Styles d'évolution dans les architectures logicielles

Olivier Le Goaer

École Centrale de Nantes Université de Nantes École des Mines de Nantes

ÉCOLE DOCTORALE STIM
« SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DES MATÉRIAUX »

Année 2009

N° attribué par la bibliothèque

Styles d'évolution dans les architectures logicielles

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE NANTES

Discipline : INFORMATIQUE

présentée et soutenue publiquement par

Olivier LE GOAER

le 9 Octobre 2009

au LINA

devant le jury ci-dessous

Président :	Pr. Laurent GRANVILLIERS	Université de Nantes
Rapporteurs :	Philippe ANIORTÉ, Professeur	Université de Pau
	Henri BASSON, Professeur	Université du Littoral
Examinateurs:	Benoit BAUDRY, Chargé de Recherche	INRIA Rennes I
	Mourad OUSSALAH, Professeur	Université de Nantes
	Abdelhak SERIAI, Maître de conférences	Université de Montpellier
	Dalila TAMZALIT, Maître de conférences	Université de Nantes

Directeur de thèse : Pr. Mourad-Chabane OUSSALAH
Encadrants de thèse : Dr. Dalila TAMZALIT et Dr. Abdelhak SERIAI
Laboratoire : Laboratoire d'Informatique de Nantes Atlantique

N° ED 503-060

STYLES D'ÉVOLUTION DANS LES ARCHITECTURES LOGICIELLES

Evolution styles within software architectures

Olivier Le Goaer

favet neptunus eunti

Université de Nantes

Olivier Le Goaer

Styles d'évolution dans les architectures logicielles

xvi+156 p.

Ce document a été préparé avec LaTeX2$_\varepsilon$ et la classe `these-IRIN` version 0.92 de l'association de jeunes chercheurs en informatique LOGIN, Université de Nantes. La classe `these-IRIN` est disponible à l'adresse :

`http://login.irin.sciences.univ-nantes.fr/`

Impression : these.tex – 12/11/2009 – 13:42

Révision pour la classe : $Id: these-IRIN.cls,v 1.3 2000/11/19 18:30:42 fred Exp

Remerciements

Au terme de ces quatre années de thèse passées au LINA, c'est avec une immense joie que j'écris ces quelques lignes de remerciements.

Je commencerais par remercier Mourad Oussalah, mon directeur de thèse, pour m'avoir donné la chance de faire une thèse sous sa direction. Il a cru en mes capacités, et il m'a donné les opportunités pour développer et prouver mes compétences. J'ai souvent eu du mal à croire en mon travail, ayant toujours le sentiment que je pouvais en faire plus. Le recul me donne à présent la mesure de ce que j'ai pu accomplir sous sa direction. J'aimerais remercier tout aussi cordialement mes encadrant(e)s de thèse, Dalila Tamzalit et Abdelhak Seriai. Ils m'ont apporté tous les conseils et les encouragements dont a besoin un doctorant pour achever sa formation à la recherche dans les meilleures conditions qui soient. Leurs conseils et leur vigilance m'ont accompagné tout au long de cette aventure.

J'aimerais exprimer ma gratitude à Messieurs Philippe Aniorté et Henri Basson pour avoir consacré un temps précieux à la lecture détaillée de mes travaux, et à Messieurs Benoit Baudry et Laurent Granvilliers pour m'avoir fait l'honneur de participer à mon jury de soutenance.

D'autre part, rien n'aurait été possible sans l'effervescence intellectuelle et amicale qui anime le LINA. C'est ainsi que tout naturellement, je remercie non seulement l'ensemble des chercheurs seniors, des doctorants, mais aussi l'ensemble des personnels du laboratoire qui ont apporté, directement ou indirectement, leur part de contribution à ce manuscrit, par leurs compétences, leur disponibilité et leur gentillesse.

J'associe à ces remerciements les membres de l'IUT de Nantes pour l'ambiance chaleureuse et pour les nombreux échanges passionnés et passionnants en « salle des profs ». J'ai trouvé dans cet établissement toutes les conditions pour m'épanouir dans le métier d'enseignant.

Sur un plan plus personnel, je tiens à remercier mes parents, pour leur soutien sans faille et leur aide. Un remerciement particulier à Sophie, qui m'a toujours soutenu et qui m'a permis de tenir le cap jusqu'au bout, même dans les moments difficiles.

J'ai énormément appris pendant ces quatre ans, tant sur le plan scientifique que sur l'organisation de la vie d'un chercheur, la manière d'écrire une publication et de valoriser son travail. Ce manuscrit de thèse marque la fin d'un chemin sinueux mais n'est que le commencement d'une route encore plus longue ...

Sommaire

Introduction

Cadre de la thèse

L'objectif de l'architecture logicielle est d'offrir une vue d'ensemble et un fort niveau d'abstraction afin d'être en mesure d'appréhender un système logiciel [GS93]. L'architecture propose une organisation grossière du système comme une collection de pièces logicielles [PW92]. Clairement, la discipline de l'architecture logicielle a favorisé la modélisation de systèmes logiciels de plus en plus complexes. Cet accroissement de la complexité s'est vu tout naturellement accompagné de questions cruciales sur l'évolution des architectures. L'évolution constitue l'aspect "vie" d'un système. Ce dernier n'est pas figé dans le temps : il évolue. Si on porte cette remarque à sa conclusion logique, l'architecture du système doit aussi évoluer, s'adapter. L'évolution traitée dans ce travail vise spécifiquement les changements de la structure de systèmes complexes à l'échelle de leur architecture. On parlera ainsi d'*évolution architecturale*. Cependant, l'utilisation de l'architecture logicielle dans le développement d'applications "grandeur réelle" a révélé rapidement des interrogations : est-il possible de faire évoluer l'architecture selon des règles ? existe-t-il des meilleures pratiques ? comment ne pas réinventer la roue à chaque fois ? En effet, lorsqu'un architecte fait face à un besoin d'évolution, il est probable que des situations similaires aient été résolues dans d'autres cas. Cette vision suggère que l'on peut identifier des évolutions réutilisables d'une architecture à une autre. Une telle approche de l'évolution doit permettre de :

- identifier le problème d'évolution à résoudre par capitalisation et organisation d'expériences,
- proposer une solution possible et normalement correcte pour y répondre,
- offrir les moyens d'exploiter cette solution dans un contexte précis.

Dans le cadre de cette thèse, nous nous intéressons spécifiquement aux architectures logicielles à base de composants [GS93, BCK98]. Ces dernières reposent sur une décomposition du système en un ensemble de composants (unités de calcul ou de stockage) qui communiquent entre eux par l'intermédiaire de connecteurs (unités d'interaction). Pour asseoir le développement de telles architectures, il est nécessaire de disposer de notations formelles et d'outils d'analyse de spécifications architecturales. Les langages de description d'architecture (en anglais, les ADLs : *Architecture Description Languages*) constituent une bonne réponse. La communauté académique a proposé ces dernières années un nombre important d'ADLs pour décrire et raisonner sur les architectures logicielles.

Notre travail consiste à explorer la notion de réutilisation pour l'évolution d'architectures logicielles décrites à l'aide des ADLs. Le métier d'architecte est au cœur de notre problématique puisque c'est son expérience et sa compétence qui alimente le processus de réutilisation. Pour capitaliser cette expérience, nous proposons le concept de *style d'évolution*. Un style d'évolution peut être vu comme une solution réutilisable à un problème d'évolution bien identifié, et ré-applicable à différentes situations. En abordant ainsi le problème de l'évolution dans les architectures à base de composants à travers l'exploitation de l'expérience passée des architectes, cette thèse se place au carrefour de l'ingénierie du logiciel et de l'ingénierie de la connaissance.

Problématique abordée

Les systèmes logiciels nécessitant une évolution adoptent généralement des approches ad-hoc et spécifiques à l'application considérée. Ces systèmes devraient bénéficier d'une approche d'évolution systématique, fondée sur des principes, et supportée par une infrastructure réutilisable [OMT98]. En d'autres termes, l'évolution doit être une discipline à part entière, munie de ses concepts, ses meilleurs pratiques, ses méthodologies et ses outils. L'évolution d'un système ne doit pas être empirique mais doit suivre des préceptes d'ingénierie, à l'image de ce qui se fait déjà pour la construction du système lui-même. C'est presque un corollaire de dire que la difficulté et les coûts faramineux des activités d'évolution sont dûs essentiellement au manque de modèles et de démarches proposés, c'est-à-dire au manque de cadre méthodologique. Pour tenter de répondre à ces défis, l'évolution est devenue un domaine de recherche à part entière, aujourd'hui bien accepté dans l'ingénierie logicielle. On trouve de nombreuse conférences et ateliers dédiés à ce sujet. Bien que tous ces efforts de recherche se situent dans le champ de l'évolution, les approches retenues pour aborder le problème de l'évolution logicielle divergent.

De manière générale, l'accroissement de la complexité dans les systèmes logiciels a nécessité une prise de recul face aux éléments en présence et fait apparaître un fort besoin d'abstraction pour pouvoir raisonner de manière correcte, sans se noyer sous les détails. En toute logique, l'évolution logicielle devrait bénéficier du même traitement afin de supporter une meilleure compréhension et une meilleure communication de cette notion. Le problème fondamental est que la connaissance nécessaire pour mener à terme une évolution, quand elle existe, reste cantonnée à l'univers du mental. En effet, une meilleure compréhension des exigences et de meilleures méthodes de modélisation et de conception nous permettent de faire évoluer un système. En partant de ce constat, notre idée directrice est de proposer un formalisme capable de recueillir le résultat de l'extraction de ce type de connaissance. L'acte de formalisation est nécessaire pour pouvoir capitaliser l'évolution et in-fine la réutiliser. La connaissance ainsi extraite peut être mise à disposition, pour soi ou pour d'autres, en vue d'accélérer et d'améliorer la qualité des activités d'évolution logicielle.

D'une part, notre approche suggère l'existence de sorte de "patterns" pour l'évolution, applicables à des familles de systèmes logiciels partageant des caractéristiques communes. En effet, nous pensons que l'évolution logicielle doit se focaliser sur des familles de logiciels plutôt que sur des produits logiciels individuels. Ceci requiert de penser en termes de familles d'évolutions plutôt qu'en termes d'évolutions singulières. D'autre part, nous considérons que l'évolution d'un système doit être abordée à un haut-niveau d'abstraction. Notre vision est enracinée dans l'argument avancé par Jazayeri – « Un système logiciel est plus que du code source ; les modèles et méta-modèles sont importants pour l'évolution logicielle » [Jaz05]. Il faut bien dire que cette approche va à contre courant des pratiques industrielles centrées sur le code, considéré comme étant le seul représentant fiable du logiciel et donc le seul artefact digne de focaliser les efforts pour lui permettre d'évoluer. En ce qui nous concerne, nous sommes convaincus que les architectures logicielles à base de composants offrent des fondations saines pour aborder le problème de l'évolution logicielle. Dans ce scénario, les responsabilités du métier d'architecte logiciel sont élargies puisque des compétences relatives à l'évolution sont exigées.

La problématique de cette thèse consiste donc à proposer aux architectes *une méthode d'ingénierie pour l'évolution architecturale*. La composante principale dans une méthode d'ingénierie est la notion de *modèle* ; la deuxième composante essentielle est la notion de *démarche*. Un modèle d'évolution puissant doit offrir à l'architecte les éléments nécessaires à la modélisation de

sa connaissance des concepts, des règles, des relations, etc., qui entrent en jeu dans le processus d'évolution. Il doit aussi pouvoir utiliser efficacement le savoir-faire qui a été acquis et raisonner avec lui pour résoudre des problèmes d'évolution et pour acquérir de nouvelles connaissances. Finalement, le modèle d'évolution doit être capable d'expliquer le raisonnement suivi pour achever une tâche d'évolution. La démarche doit venir en accompagnement du modèle, en tant que processus grâce auquel s'effectue le travail de modélisation. Ces deux aspects, modèle et démarche, constituent les contributions principales de cette thèse, présentées ci-après.

Contributions

La première contribution que nous proposons est le modèle d'évolution SAEM (Style-based Architectural Evolution Model), reposant sur la modélisation à base de styles pour décrire des évolutions logicielles récurrentes. La nouveauté principale de ce modèle est de proposer le concept de *style d'évolution*, doté de qualités descriptive et prescriptive de l'évolution. La première qualité sert à véhiculer des connaissances et partager un vocabulaire d'évolution commun entre les architectes. La seconde qualité sert à guider l'architecte dans son activité d'évolution, en contraignant l'espace des solutions possibles. Aussi, le modèle SAEM proposé supporte la réutilisabilité en augmentant l'extensibilité, l'évolutivité et la compositionnalité des évolutions modélisées à base de styles. Nous proposons un formalisme de description des concepts des styles d'évolution baptisé "modèle en Y". Ce modèle a pour socle un tryptique représentant trois aspects importants : le domaine, l'entête et la compétence. Ces aspects représentent successivement tout ce qui est lié aux éléments évolutifs d'une architecture, aux opérations d'évolution sur ces éléments et aux méthodes utilisées pour achever ces opérations.

La seconde contribution concerne une démarche d'élaboration de bibliothèques pour les styles d'évolution, avec pour objectif de remplacer les évolutions que l'on cherche à développer *ex nihilo* par une réutilisation de l'existant. Cette démarche inclut différents niveaux d'intervention correspondants à des acteurs distincts. Le modèle MY unifie les concepts du paradigme des styles d'évolution en supportant trois niveaux de modélisation : méta-style d'évolution, type de style d'évolution, et instance de style d'évolution. Ce modèle améliore la réutilisation des styles d'évolution en supportant une bibliothèque multi-hiérarchique pour les trois niveaux de modélisation. Une infrastructure de réutilisation est bâtie spécifiquement au dessus de la bibliothèque de niveau type afin d'en contrôler le peuplement et l'interrogation par les architectes, selon que l'on se place du point de vue du fournisseur ou du consommateur d'évolutions. Compte tenu de la structure hiérarchique de la bibliothèque, le fonctionnement de l'infrastructure s'appuie sur des raisonnements de nature classificatoire.

Organisation du document

Le corps de ce manuscrit est organisé comme suit :

Le chapitre 1 pose le lit de notre travail et propose un état du domaine de l'architecture logicielle, notamment dans ses aspects nécessaires à notre travail. Le chapitre débute sur une rétrospective de l'architecture logicielle, depuis les premières intuitions des années 70 jusqu'à son éclosion en tant que discipline dans les années 90. Malgré le gain en popularité de l'architecture

logicielle ces dix dernière années, le vocabulaire est souvent mal utilisé et les objectifs mal compris. Ainsi, il est utile de rappeler quelques définitions notables de l'architecture logicielle proposées dans la littérature et de revenir sur la notion de style architectural qui a profondément influencé notre travail. Le chapitre se poursuit avec les travaux menés sur les langages de description d'architectures logicielles (les ADLs). Le chapitre se termine par l'emploi d'une technique de méta-modélisation pour la description d'une architecture logicielle, qui sera à prendre en compte lorsque la problématique de l'évolution sera abordée.

Le chapitre 2 cherche à situer le contexte du travail à travers notre motivation et les avantages à traiter l'évolution à un haut-niveau d'abstraction comme celui offert par les architectures logicielles. Puis, nous proposons de caractériser ce que l'on peut attendre d'un modèle d'évolution à l'aide d'une pyramide de besoins. De la base au sommet de cette pyramide, on trouve différentes préoccupations d'importance décroissante. Muni de ce cadre de comparaison, nous étudions neuf modèles d'évolution représentatifs des écoles académiques américaine, européenne et française. De cette étude nous faisons ressortir un certain nombre de limitations qui vont nous permettre de guider la proposition de notre propre modèle d'évolution.

Le chapitre 3 décrit notre modèle d'évolution en réponse aux limitations identifiées dans les approches étudiées. Ce modèle baptisé SAEM a pour objectif d'aborder le problème de l'évolution architecturale sur la base du concept de style. Les styles d'évolution sont des citoyens de première classe, pouvant ainsi être manipulés. Ils sont comparables à des patterns pour l'évolution, spécifiés une seule fois, mais réutilisables à l'infini sur des architectures logicielles. Le cœur de SAEM est présenté sous la forme d'un méta-modèle dont les concepts et les mécanismes sont détaillés indépendamment de tout ADL. Nous introduisons un formalisme de représentation des styles d'évolution baptisé MY. Ce formalisme a pour socle une représentation triptyque formant les branches d'un 'Y'. Nous concluons ce chapitre en évaluant notre propre modèle d'évolution SAEM à travers son positionnement dans la pyramide des besoins proposée au chapitre précédent.

Le chapitre 4 fait état d'une démarche pour guider l'architecte dans son processus de modélisation de l'évolution selon SAEM, ainsi que de la réutilisation des styles d'évolution. Sur la base du formalisme MY, nous expliquons comment il est possible de construire des bibliothèques pour les styles d'évolution. Ces bibliothèques stockent des éléments réutilisables à différents niveaux de modélisation dans l'objectif commun d'une approche pour et par la réutilisation, qui sont les deux faces d'une même pièce. Nous décrivons une infrastructure de réutilisation capable d'exploiter les styles d'évolution ainsi capitalisés. Le cœur opérationnel de l'infrastructure repose sur deux techniques de raisonnement par classification.

Le chapitre 5 vise à projeter SAEM spécifiquement sur COSA, un ADL développé par notre équipe et mixant des mécanismes objets avec des concepts des architectures logicielles. COSA s'avère être une structure d'accueil idéale pour SAEM puisque tous deux partagent des mécanismes opérationnels communs. La finalité de cette projection est d'assurer l'implémentation des styles d'évolution sur une plateforme technologique cible. Puis, nous menons une expérimentation de COSA et de SAEM dans le cadre du projet industriel ZOOM. Ce projet vise la conception et l'évolution d'architectures de systèmes navals complexes. L'expérimentation se concentre sur les réseaux de tuyauterie, où un consultant expert métier est intervenu pour nous assister.

Plusieurs annexes sont également fournies en fin de document.

L'annexe 1 propose une étude complémentaire de trois référentiels issus de la littérature et dédiés à l'évolution basée architecture.

L'annexe 2 présente les algorithmes utilisés pour mettre en œuvre un raisonnement classificatoire. Ces algorithmes sont simples mais méritent d'être mentionnés.

L'annexe 3 revient sur le développement de l'outil COSAStudio et donne un aperçu des choix et solutions techniques retenues.

Nous résumons la structuration de la thèse dans le schéma 1.1.

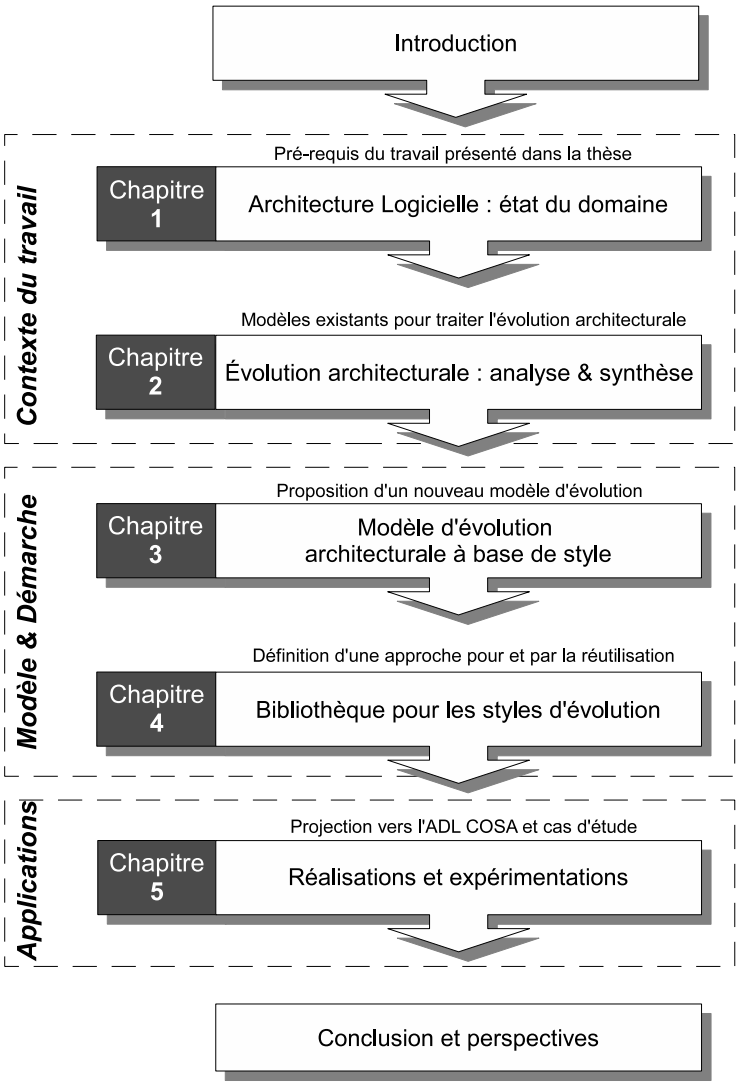

Figure 1.1 – Représentation schématique de la structure de cette thèse.

Liste des publications obtenues

Conférences internationales avec comité de lecture et Actes

■ Olivier Le Goaer, Dalila Tamzalit, Mourad Oussalah, and Abdelhak Seriai. Evolution Shelf : Reusing Evolution Expertise within Component-Based Software Architectures. In *The 32nd Annual IEEE International Computer Software and Applications Conference* , pages 311–318, Turku, Finland, July 2008.

■ Olivier Le Goaer, Mourad Oussalah, Dalila Tamzalit and Abdelhak Seriai. Evolution Shelf : Exploiting Evolution Styles within Software Architectures. In *20th International Conference on Software Engineering and Knowledge Engineering* , pages 387–392, San Francisco Bay, USA, July 2008.

■ Olivier Le Goaer, Mourad Oussalah, Dalila Tamzalit, and Djamel Serai. Evolution Styles in Practice - Refactoring Revisited As Evolution Style. In *The Second International Conference on Software and Data Technologies*, volume 3, pages 138–143, Barcelona, Spain, July 2007. INSTICC Press.

■ Mourad Oussalah, Dalila Tamzalit, Olivier Le Goaer, and Abdelhak Seriai. Updating styles challenge updating needs within component-based software architectures. In *The Eighteenth International Conference on Software Engineering and Knowledge Engineering*, pages 98–101, San Francisco Bay, USA, July 2006.

■ Abdelhak Seriai, Mourad Chabane Oussalah, Dalila Tamzalit, and Olivier Le Goaer. A reuse-driven approach to update component-based software architectures. In *The 2006 IEEE International Conference on Information Reuse and Integration*, pages 313–318, Waikoloa, Hawaii, USA, September 2006.

■ Dalila Tamzalit, Mourad Oussalah, Olivier Le Goaer, and Abdelhak Seriai. Updating software architectures : A style-based approach. In *The 2006 International Conference on Software Engineering Research and Practice*, pages 336–342, Las Vegas, Nevada, USA, June 2006.

■ Olivier Le Goaer, Dalila Tamzalit, Mourad Oussalah, and Abdelhak Seriai. How to manage update needs within component-based software architectures. In *The 32nd Euromicro Conference on Software Engineering and Advanced Applications (Work In Progress Session)*, pages 8–10, Dubrovnik, Croatia, August 2006.

Workshops Internationaux avec Comité de Lecture et Actes

■ Olivier Le Goaer, Dalila Tamzalit and Mourad Chabane Oussalah, and Abdelhak-Djamel Seriai. Evolution styles to the rescue of architectural evolution knowledge. In *Proceedings of the 3rd international workshop on Sharing and reusing architectural knowledge*, pages 31–36, Leipzig, Germany, May 2008.

■ Olivier Le Goaer and Peter Ebraert Evolution styles : change patterns for Software Evolution

In *Proceedings Third International ERCIM Workshop on Software Evolution*, pages 252–261, Paris, France, October 2007.

■ Abdelkrim Amirat, Adel Smeda, and Olivier Le Goaer. Integrating Aspects in the COSA Model. In *The Eighteenth International Symposium on Programming and Systems*, pages 79–87, Alger, Algérie, May 2007. Université des Sciences et de la Technologie Houari Boumediene (USTHB).

Workshops Nationaux avec Comité de Lecture sans Actes

■ Olivier Le Goaer, Mourad Oussalah, Dalila Tamzalit, and Seriai Djamel. Evolution dirigée par les styles. In *Atelier RIMEL (Rétro-Ingénierie, Maintenance et Evolution des Logiciels)*, Toulouse, France, March 2007.

Chapitre 1

Architecture logicielle : état du domaine

Le travail présenté dans cette thèse relève de l'évolution dans les architectures logicielles. Auparavant, il est nécessaire de poser clairement le domaine et lorsque nécessaire, d'éclairer certains points et caractéristiques que nous jugeons importants pour notre travail. L'objectif de ce premier chapitre est de clarifier la notion d'"Architecture Logicielle", très fréquemment rencontrée dans les travaux liés au développement des systèmes logiciels. Dans ce but, nous proposons un état du domaine d'une discipline désormais majeure qui a mûrie de manière significative ces quinze dernières années.

Un grand intérêt est porté au domaine des architectures logicielles [CGL*03]. Cet intérêt est motivé principalement par la réduction des coûts et des délais de développement des systèmes complexes. L'architecture logicielle permet d'exposer de manière compréhensible et synthétique la complexité d'un système logiciel et de faciliter l'assemblage de pièces logicielles. Elle permet en effet au concepteur de raisonner sur des propriétés (fonctionnelles et non fonctionnelles) d'un système à un haut niveau d'abstraction. L'architecture joue le rôle essentielle de passerelle entre l'expression du besoin d'un système et l'étape de codage du logiciel. Pour décrire les architectures logicielles, plusieurs langages de description (ADL : *Architecture Description Language*) ont été proposés. Les ADLs offrent un niveau d'abstraction élevé pour la spécification et le développement des système logiciels, qui se veut indépendant des langages de programmation et de leurs plateformes d'exécution.

Dans ce chapitre, nous remontons d'abord aux tournants idéologiques fondateurs de l'architecture logicielle datant du début des années 70, jusqu'à leur véritable émergence comme discipline de premier plan. A ce titre, nous étudions la place que prend désormais l'architecture dans le cycle de développement d'un système logiciel. La communauté académique a profondément influencé cette discipline, et il nous semble utile de répertorier les définitions majeures de l'architecture logicielle proposées dans la littérature. Nous nous attardons sur le concept de style architectural [SG96], considéré par beaucoup comme un élément clé de l'architecture logicielle. Nous abordons les langages de description d'architecture au travers de leurs principaux concepts descriptifs. Pour terminer, nous montrons que la spécification d'une architecture logicielle par un ADL peut passer par plusieurs niveaux de modélisation. Il s'agit là de voir l'architecture logicielle à travers le prisme de la méta-modélisation.

1.1 Les architectures logicielles

Le domaine de l'informatique est apparu à travers des problèmes liés à la complexité, et cela depuis sa création. Au début, les problèmes de complexité ont été résolus par les développeurs en

choisissant les bonnes structures de données, en développant des algorithmes, et en appliquant tacitement le concept de séparation des préoccupations. Bien que le terme "architecture logicielle" est relativement nouveau pour l'industrie, les principes fondamentaux du domaine ont été appliqués sporadiquement par les pionniers de l'ingénierie du logiciel depuis le milieu des années 1980. Les premières tentatives pour capturer et expliquer l'architecture logicielle d'un système furent imprécises et désorganisées – souvent caractérisées par un ensemble de diagrammes non-formalisés. Dans les années 1990 il y a eu un effort commun pour tenter de définir et de codifier les aspects fondamentaux de la discipline. Des ensembles préliminaires de patrons de conception, de styles, de meilleures pratiques, de langages de description et de logiques formelles ont été élaborés au cours de cette période.

1.1.1 Émergence de l'architecture logicielle

La discipline de l'architecture logicielle concerne essentiellement les systèmes à large échelle, par conséquent complexes. En effet, c'est l'accroissement de la complexité qui a amené progressivement à repenser la façon de construire les système logiciels. A l'automne 1968, lors d'une conférence de l'OTAN où 50 des meilleurs programmeurs de l'époque furent conviés, les experts en informatique ont anticipé ce qui allait devenir la fameuse “crise du logiciel”. Voici un extrait tiré de la synthèse des échanges qui ont eu lieu lors de cette conférence [NR68] et qui nous semble très important du point de vue idéologique :

> « The challenge of complexity is not only large but also growing [...]. To keep up with such demand, programmers will have to change the way that they work. You can't build skyscrapers using carpenters », Curtis quips

Dans les années 70, le secteur de l'ingénierie logicielle est en pleine mutation et doit répondre à de nouvelles exigences, du fait de l'accroissement de la taille et de la complexité des systèmes. Les moyens ne sont plus alignés sur les enjeux, et il devient nécessaire de changer l'échelle à laquelle le développement est traditionnellement considéré. Ces nouveaux défis ne pourront être relevés que par la maîtrise des différents niveaux de détails et de raisonnement pouvant être isolés dans un système logiciel.

1.1.1.1 Le défi des systèmes complexes

Dans leur article fécond intitulé “Programming-in-the-large versus programming-in-the-small” [DK75], DeRemer *et al.* insistent sur le tournant qui doit être amorcé dans le développement logiciel en discutant de la transition du développement classique à petite échelle vers un développement à plus grande échelle. Leur langage MIL75 est un précurseur de la famille des langages d'interconnexion de modules (*Module Interconnection Languages* – MILs), conçus pour faire tenir ensemble des modules, c'est-à-dire des segments de programmes. Un langage d'interconnexion de modules doit permettre d'identifier les modules du système et indiquer comment ils s'intègrent ensemble pour mettre en œuvre les fonctionnalités du système, mais n'est pas concerné par ce que le système fait ni comment les modules implantent individuellement leurs fonctionnalités [PDN86]. Les MILs prônent donc un glissement de paradigme : celui du développement à base de lignes de codes vers le développement à base de modules. Cette rupture avec les méthodes existantes nécessite une programmation à haut niveau, à savoir, l'assemblage d'éléments réutilisables développés indépendamment (en C, en Ada, etc.). Ainsi, la programmation à large échelle

fait intervenir des capacités d'abstraction (jusqu'à ce qu'un module soit implémenté, il demeure une abstraction) et des capacités de gestion car la construction d'abstractions ne se limite pas seulement à décrire des choses qui peuvent fonctionner mais également gérer les efforts – humains – qui les feront fonctionner.

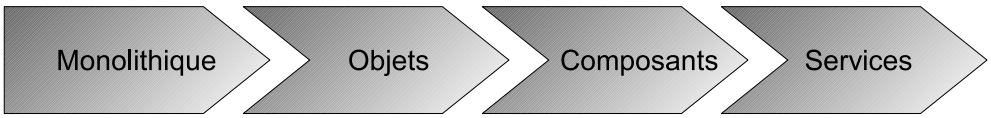

Figure 1.1 – Catégories d'architecture logicielle face à la complexité croissante des systèmes.

Le constat clé est que plus un système logiciel devient complexe, plus il devient nécessaire d'expliciter sa structure. Face à l'accroissement de la complexité, la vision de l'ingénierie des logiciels s'est donc radicalement modifiée en passant de la perspective de développement d'un système monolithique – ou "monobloc" – vers un système construit comme un assemblage de morceaux logiciels. Ainsi, de nouvelles catégories d'architectures ont émergées, offrant chacune des unités granulaires différentes pour le partitionnement d'un système (cf. figure 1.1). Les objets constituent des briques de base qui ont l'avantage de trouver une correspondance directe avec les langages de programmation à objet. L'image retenue est celle d'un ensemble d'objets qui communiquent entre eux en s'échangeant des messages afin de collaborer dans un objectif commun. Cependant, même si l'orienté objet offre une base saine pour le développement d'éléments réutilisables à fine ou à plus forte granularité [Jaz95], elle n'offre pas les notations adéquates pour la construction à plus large échelle. Les composants constituent des briques de construction à plus large échelle et sont vus comme le prolongement des objets [Szy98]. Traditionnellement, les composants sont à "gros grain" [SG96] car ils offrent de nombreuses fonctionnalités et/ou des fonctionnalités complexes. L'architecture à base de composants vise à construire un système à la manière d'un puzzle, où chaque composant est une pièce bien définie et destinée à être assemblée à une tierce pièce. L'emboîtement des pièces repose sur la correspondance entre les fonctionnalités qu'elles offrent et celles qu'elles requièrent. Puis, la forte complexité de certains systèmes en termes d'hétérogénéité dans les technologies utilisées, et en terme de distribution à travers les réseaux a fait naître les architectures à base de services. Les services constituent des briques de construction à très large échelle destinées à construire des systèmes logiciels caractérisés par un environnement distribué et hautement dynamique. Ici, la métaphore n'est plus celle du puzzle, mais celle d'un ensemble de fournisseurs et de consommateurs de services qui a priori ne se connaissent pas, mais qui ont vocation à être mis en relation dans un cadre contractuel.

Il est important de mentionner que ces catégories d'architectures forment des couches conceptuelles successives. On peut concevoir un service (*e.g.*, traduction de documents accessible par le Web) reposant sur des composants (*e.g.*, moteur de traduction, filtrage, base de définitions), eux-mêmes reposant sur des objets. Chaque catégorie d'architecture offre une couche d'abstraction sur la précédente permettant de masquer progressivement les détails pour se focaliser sur des objectifs proportionnels et sur des préoccupations relatives à la vue que l'on a sur le système.

1.1.1.2 De la nécessité de décomposer

L'idée basique est qu'à défaut de réduire la complexité il faut la maîtriser en donnant une illusion de simplicité. Cela implique d'appliquer des critères de partionnements, comme le souligne David Parnas dans [Par72]. C'est ici qu'intervient la notion de décomposition afin de mettre en

pratique le célèbre adage “Diviser pour mieux régner”. Le rôle de la décomposition est d’opérer un raffinage récursif jusqu’à obtention d’éléments compréhensibles. Toutefois, la notion d’“élément compréhensible” dépend totalement du système considéré et le nombre raffinages requis n’est pas connu a priori. Chaque étape de raffinage correspond à un niveau de décomposition hiérarchique. On emploi souvent le terme de "granularité" pour se référer à un niveau plus ou moins profond dans cette hiérarchie. On parle alors de granularité forte ou fine. La décomposition hiérarchique est fondamentalement une stratégie "top-down". D’abord le système est divisé en sous-systèmes de premier niveau. Puis, chaque sous-système de premier niveau est divisé en sous-systèmes de second niveau, et ainsi de suite. La décomposition hiérarchique permet d’aller à différents niveaux de profondeurs dans les sous-systèmes sans perdre la vue d’ensemble du système. Les sous-systèmes des niveaux supérieurs contiennent tous les détails des sous-systèmes des niveaux inférieurs.

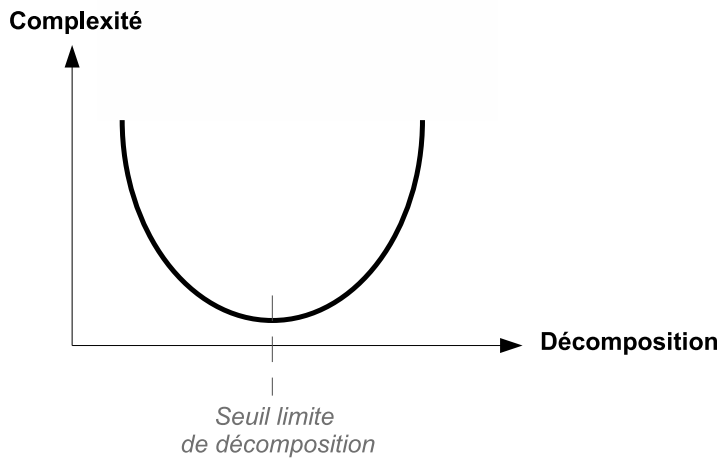

Figure 1.2 – Influence de la décomposition sur la complexité d’un système.

Le travers de cette activité est de réintroduire une autre forme de complexité. En effet, la décomposition d’un système en sous-systèmes implique l’existence de liaisons entre les sous-systèmes. Ce sont ces liaisons qui maintiennent les “parties” ensembles et les font collaborer pour former un “tout”. Par conséquent, une trop forte décomposition augmente dramatiquement le nombre de liaisons internes et les efforts nécessaires pour les gérer. Nous illustrons ce principe important par la métaphore du $\cup$ (cf. figure 1.2). Selon cette métaphore, la base du $\cup$ délimite le degré de décomposition maximal acceptable afin de ne pas ré-augmenter la complexité du système.

1.1.2 Reconnaissance de l’architecture logicielle

Le numéro spécial de l’été 2006 de l’*IEEE Software magazine* a été dédié au domaine de l’architecture logicielle pour commémorer deux décennies de recherche et une décennie de pratique. On s’accorde à penser que l’architecture logicielle a suffisamment mûrie pour être aujourd’hui

unanimement reconnue comme une discipline majeure [SC06, KOS06], et que la pratique de l'architecture logicielle a été élevée au rang de spécialité à part entière du génie logiciel. Notamment, l'intégration de l'architecture logicielle dans le cycle de développement des logiciels est jugée indispensable sous peine de faire échouer les projets. C'est en substance ce que dit la définition assez originale de Eoin Woods :

> « Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled. » - Eoin Woods

En quelque sorte, négliger l'architecture est un *anti-pattern* dans le processus de développement du logiciel. On admet aujourd'hui qu'une "bonne" architecture a une influence positive sur la qualité du produit final et qu'à l'inverse, une "mauvaise" architecture peut avoir des conséquences désastreuses, jusqu'à l'arrêt du projet [Gar00b].

1.1.2.1 L'architecture au coeur du développement

Un système est le résultat d'innombrables décisions individuelles des développeurs, et au bout d'un certain temps et d'une certaine taille, on se rend compte que l'on dispose d'une architecture. En fait, toute implantation d'un système logiciel possède nécessairement une architecture, mais cette dernière demeure totalement implicite. L'avantage du développement centré architecture est de décrire explicitement l'architecture. Cela requière de voir l'architecture logicielle comme une phase du cycle de développement à part entière [1].

Traditionnellement, le processus de développement logiciel a été modélisé sous formes d'étapes incrémentales mettant en jeu le raffinement, démarrant sur une évaluation du problème à résoudre et aboutissant avec l'implémentation du système. Une itération entre les étapes est prévue. Par exemple, pendant la phase d'implémentation, on peut découvrir des erreurs qui exigent un changement majeur dans la conception détaillée du système. Cela nécessite un retour sur l'étape de conception détaillée et une répercution sur les étapes suivantes. Une architecture explicite et capturée formellement offre un médium pour analyser très tôt l'exactitude et la complétude de la conception censée satisfaire les exigences du systèmes. La Figure 1.3 illustre un modèle de développement reconnaissant pleinement l'architecture. Ainsi, on trouve les quatre étapes suivantes :

- L'analyse des besoins vise à déterminer l'information et le traitement de cette information.
- La conception architecturale concerne la sélection des éléments architecturaux, de leurs interactions, en précisant les contraintes qui pèsent sur ces éléments et leurs interactions. L'architecture fournit un cadre qui satisfait aux besoins et sert de base pour la conception.
- La conception détaillée traite de la modularisation, de la spécification détaillée des interfaces des éléments de conception, leurs algorithmes et procédures, et les types de données nécessaires pour soutenir l'architecture. La conception doit satisfaire les propriétés spécifiées à la fois dans l'architecture et par les besoins.
- L'implémentation est liée aux représentations des algorithmes et des types de données. L'implémentation doit satisfaire la conception, l'architecture, et les besoins.

1. Néanmoins, certains auteurs considèrent toujours l'architecture logicielle non pas comme un phase au sens strict, mais plutôt comme une discipline qui imprègne toutes les phases du cycle de développement.

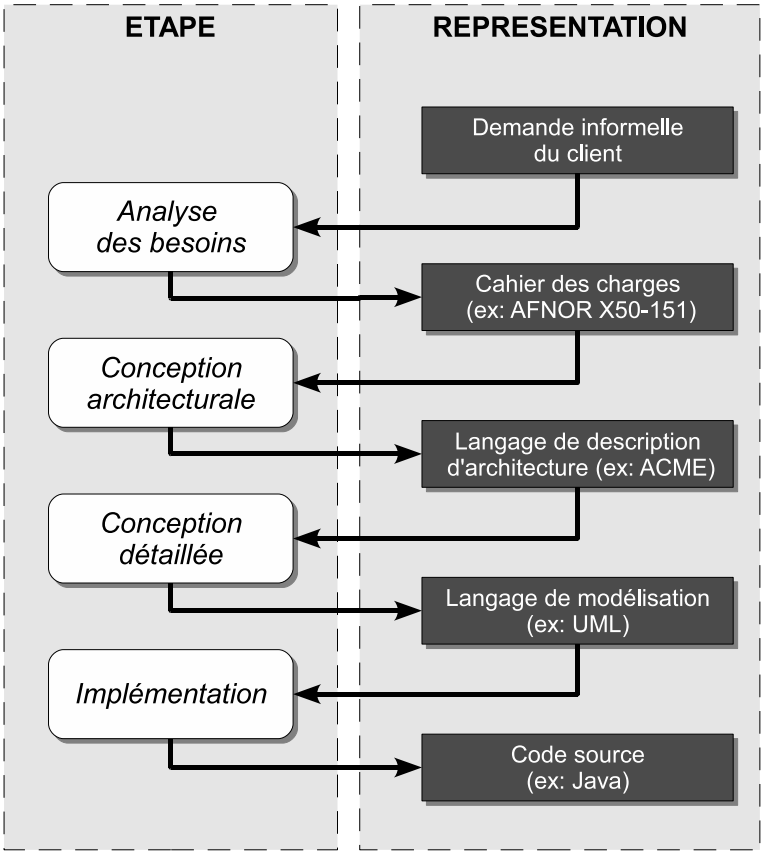

Figure 1.3 – Étapes d'un modèle de développement en cascade et leurs représentations.

1.1.2.2 Les acteurs engagés dans le développement

A chaque étape du modèle de développement correspond un acteur, possédant des aptitudes spécifiques. Le développement centré architecture rajoute un nouvel acteur, l'*architecte*, aux acteurs plus traditionnels de concepteur et de programmeur. Ainsi, l'architecte, le concepteur et le programmeur sont les trois acteurs qui participent respectivement à la phase de conception architecturale, de conception détaillée et d'implémentation. Tous trois peuvent manipuler des concepts similaires, seul le niveau auquel ils sont manipulés varie.

Tandis que le rôle d'un concepteur et d'un programmeur est plutôt bien établi, l'architecte logiciel est un terme général qui peut faire référence à un large éventail de rôles. Il existe de nombreuses définitions acceptées. La place d'un architecte logiciel dans les projets informatique a été intensivement discuté dans [Fow03] et [SS01]. Nous proposons ici nos propres définitions :

- Le programmeur : il est issu de la communauté des composants logiciels. Son travail vise à fournir des briques de base de qualité, réutilisables pour la production d'applications, qu'il met à disposition à travers des librairies. L'accent est mis sur la manière de définir un composant avec les propriétés voulues et bien agencées. La partie relative à l'assemblage de ces composants est souvent assez pauvre. Le programmeur utilise le formalisme des langages de programmation dans des environnements intégrant la notion de composant (COM/DCOM, DLL, CORBA, Java Bean, OpenCom, OSGi, etc.).
- Le concepteur : il est issu de la communauté des composants conceptuels. Il joue un rôle tampon entre le programmeur, travaillant à un faible niveau d'abstraction, et l'architecte, travaillant à un haut niveau d'abstraction. Son travail vise à définir des briques conceptuelles qui satisfont chaque problème et réutilisables pour d'autres problème similaires, qu'il met à disposition sous formes de patrons ou de frameworks.
- L'architecte : il est issu de la communauté des composants architecturaux. Son travail vise à l'intégration de ces briques à un niveau plus global pour construire des applications de qualité, maintenables. Contrairement au programmeur, la façon d'obtenir les composants est secondaire, alors que les propriétés de leurs interactions sont étudiées en détail. Il utilise le formalisme des ADLs. Des outils peuvent l'aider à analyser certaines propriétés de son architecture.

1.1.3 Définitions notables

Le terme d'"Architecture Logicielle" est probablement l'un des plus surexploité et des plus surchargé en génie logiciel. Au moins deux raisons expliquent ceci. La première est que le seul mot "architecture" apporte avec lui un certain nombre de significations implicites et d'hypothèses de par son association directe avec la construction physique de bâtiments. La deuxième raison qui entretient la confusion est que l'architecture logicielle n'est reconnue comme un produit à part entière du cycle de développement logiciel que depuis quelques années, et il aura donc fallu attendre la fin des années 80 pour que la notion d'"architecture logicielle" émerge [Sha89]. Depuis lors, l'utilisation du terme a grimpé en flèche, très souvent avec des acceptions différentes. Par exemple, une architecture logicielle a été diversement définie comme un prototype exécutable [RR91], une description statique d'une topologie [GS93], ou encore une documentation [Kru95].

A l'heure actuelle, il n'existe toujours pas de définition précise et normalisée de l'architecture logicielle, et il en existe de nombreuses interprétations. A titre d'exemple, le SEI (*Software Engineering Institute*) a documenté et catalogué des centaines de définitions [www]. Dans la

section suivante, nous nous contenterons de présenter les définitions les plus influentes, classées par ordre chronologique.

1.1.3.1 Perry et Wolf (1992)

L'une des premières définitions formelles d'architectures logicielles, de Perry et Wolf [PW92], est restée l'une des plus perspicaces. Après avoir examiné les architectures dans d'autres disciplines (matériel, réseaux et bâtiments), Perry et Wolf décrivent une architecture logicielle comme "un ensemble d'éléments architecturaux qui ont une forme particulière". Les éléments sont divisés en trois catégories : éléments de traitement, éléments de donnée, et éléments de connexion. Étant donné que les éléments de traitement et de donnée ont été été largement étudiés par le passé (par exemple, les fonctions et les objets), ce sont les éléments de connexion qui distinguent une architecture d'une autre. La forme de l'architecture est donnée en énumérant les propriétés des différents éléments et les relations entre ces éléments. Un autre élément essentiel de l'architecture est sa logique (*rationale* en anglais) qui englobe, entre autres, les aspects de qualité.

Perry et Wolf définissent également un style architectural comme "ce qui abstrait les éléments et les aspects formels de diverses architectures spécifiques". En bref, un style est une architecture moins contrainte et moins complète – il n'y a pas de ligne de démarcation franche. L'intérêt d'un style particulier réside dans le fait qu'il traite des choix de conception importants dès le départ, en isolant et en mettant l'accent sur certains aspects. Un style ou une architecture spécifique peut être vue de trois façons différentes, sur la base de ses éléments : la vue traitement, la vue donnée et la vue connexion. Les trois vues sont nécessaires à la compréhension du style ou de l'architecture.

1.1.3.2 Garlan et Shaw (1993)

Un définition très populaire de l'architecture logicielle a été avancée par Garlan et Shaw [GS93], qui est plus restrictive que la définition de Perry et Wolf. Garlan et Shaw ont proposé qu'une architecture logicielle pour un système spécifique soit représentée comme "un ensemble de composants de calcul - ou tout simplement de composants - accompagné d'une description des interactions entre ces composants - les connecteurs". Sur la base de cette définition, les auteurs ont utilisé le terme de style architectural pour désigner une famille de systèmes (c'est-à-dire d'architectures applicative) qui partagent un vocabulaire commun de composants et connecteurs, et qui répondent à un ensemble de contraintes pour ce style. Les contraintes peuvent porter sur une variété de choses, notamment sur la topologie des connecteurs et des composants, ou sur la sémantique de leur exécution.

1.1.3.3 Bass, Clements et Kazman (1998)

Dans leur définition, Bass *et al.* [BCK98] insistent sur les propriétés extérieurement visibles d'un composant qui définissent son comportement attendu : elles traduisent les hypothèses que d'autres composants peuvent faire sur ce composant (par exemple, les services qu'il fournit, les ressources requises, ses performances, ses mécanismes de synchronisation). Le qualifiant "extérieurement visible" exprime le pendant des détails que le composant encapsule. Un composant est donc une unité d'abstraction dont la nature dépend de la structure considérée dans le processus de conception architecturale. Ce peut être un service, un module, une bibliothèque, un processus, une procédure, un objet, une application, etc. Mais le comportement observable de

chaque composant fait partie de l'architecture puisqu'il détermine ses interactions possibles avec d'autres composants.

La définition proposée met également en lumière le fait qu'un système peut avoir plusieurs structures correspondant chacune à un point de vue, donc à une finalité ou classe de problèmes précis à résoudre. En somme, comme en architecture du bâtiment, un logiciel est représenté par plusieurs plans et schémas destinés chacun à un corps de métier et dépendants de l'étape du processus de développement.

1.1.3.4 La norme AINSI/IEEE Std 1471 (2000)

L'IEEE a défini une norme pour la description architecturale de systèmes à prédominance logicielle (*software-intensive systems*), l'IEEE 1471 [2] [IEE00]. L'IEEE 1471 a sciemment proposé une définition généraliste capable d'englober plusieurs interprétations. Surtout, la norme IEEE 1471 établit un cadre conceptuel (cf. figure 1.4) et un vocabulaire pour parler des problématiques architecturales des systèmes. Elle comprend l'utilisation de plusieurs vues, de modèles réutilisables au sein de vues, et la relation qu'entretient l'architecture avec le contexte du système (appelé aussi environnement). En se basant sur ce cadre conceptuel, l'idée est d'identifier et d'édicter des pratiques architecturales pertinentes, et de les incuber.

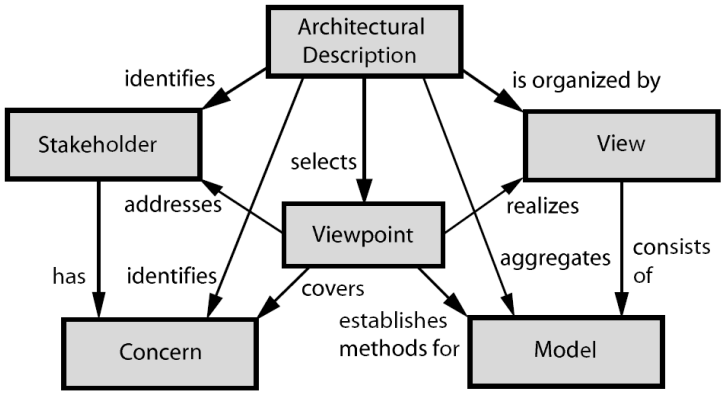

Figure 1.4 – Modèle conceptuel pour la description d'architecture logicielle (adapté de la norme 1471-2000).

Quel que soit le bien-fondé d'un ADL en particulier, il n'y a pas de consensus dans la communauté des utilisateurs de la norme IEEE 1471 sur les ADLs. Il n'y a même pas de consensus sur ce qu'un ADL devrait être. Toutefois, il y a consensus sur le fait que les architectes doivent choisir leurs notations avec soin, documenter ces choix, et être encouragés à utiliser des ADLs bien définis. Chaque utilisateur de l'IEEE 1471 peut ainsi choisir un ou plusieurs ADL, ou même créer le sien dans le cadre prévu par l'IEEE 1471, qui inclue les éléments suivants :

- Description architecturale (*Architectural Description*) : un ensemble de vues et d'autres informations architecturales.
- Partie prenante (*Stakeholder*) : un individu, un groupe ou une organisation qui a au moins une préoccupation en rapport avec le système.

2. A été également adoptée comme standard ISO en 2006 et publiée en tant que ISO/IEC 42010 :2007

- Préoccupation (*Concern*) : une exigence fonctionnelle ou non-fonctionnelle.
- Vue (*View*) : un ensemble de modèles représentant un système selon un ensemble de préoccupations connexes.
- Point de vue (*Viewpoint*) : les conventions pour la création, la représentation et l'analyse d'une vue.
- Modèle (*Model*) : un diagramme ou une description construits suivant la méthode définie dans un point de vue. Ces derniers fournissent la description spécifique du système, qui peut inclure des sous-systèmes.

1.1.4 Styles architecturaux

En plus des structures de l'architecture logicielle, les systèmes ont souvent en commun des patrons d'organisation, appelés *styles architecturaux* [SG96]. Certains des plus répandus qui ont été identifiés et décrits sont le style client/serveur, le style tuyau&filtre, le style orienté objet et le style en couches. L'un des avantages des styles architecturaux est qu'ils possèdent des attributs de qualité déterminés [BCK98] (*e.g.*, performance, disponibilité, sécurité, réutilisabilité, etc.). Cela permet à un architecte de sélectionner un style en fonction des exigences requises par le système au lieu d'inventer une architecture à partir de zéro [BCK98]. Dans la mesure où les styles architecturaux peuvent traiter différents aspects de l'architecture logicielle, une architecture donnée peut être issue de plusieurs styles.

À notre avis, les patrons et les styles architecturaux sont de bon mécanismes pour encapsuler de l'expertise pour la conception. L'introduction de la notion de réutilisation dans les architectures logicielle est un point remarquable, dont les bénéfices sont aujourd'hui reconnus. D'une part, un style architectural partage avec la notion de patron l'idée que des solutions à des problèmes de conception courants peuvent être réutilisées. D'autre part, il est fréquent de mettre en relation les styles architecturaux. Par exemple, un style hybride peut être formé en combinant plusieurs styles de base dans un style unique et bien coordonné.

1.1.4.1 Définitions

Une architecture incarne à la fois des propriétés fonctionnelles et non fonctionnelles et par conséquent il peut être difficile de comparer directement les architectures de différents types de systèmes, ou bien le même type de système dans des environnements différents. Les styles sont un moyen pour classer les architectures et pour définir leurs caractéristiques communes [NR99].

Perry et Wolf [PW92] définissent le style architectural comme une abstraction des types d'élément et des aspects formels de diverses architectures, pour se concentrer seulement sur certains aspects d'une architecture. Un style architectural encapsule d'importants choix d'éléments architecturaux et met l'accent sur les contraintes importantes imposées sur les éléments et leurs relations. Cette définition admet des styles qui se concentrent uniquement sur les connecteurs d'une architecture, ou sur des aspects spécifiques des interfaces des composants. En revanche, Garlan et Shaw [GS93], Garlan *et al.* [GAO95], et Shaw et Clements [SC97], définissent tous le style en terme de patron d'interaction entre des types de composants. En particulier, un style architectural détermine le vocabulaire des composants et des connecteurs qui peuvent être utilisés dans les instances de ce style, avec un ensemble de contraintes sur la manière dont ils peuvent être combinés [GS93]. Cette vision des styles architecturaux est le résultat direct de leur définition de l'architecture logicielle – penser une architecture comme une description formelle, plutôt

que comme un système à l'exécution –, et conduit à des abstractions fondées uniquement sur les patrons partagés par un ensemble de diagrammes de type "boîte et ligne". Abowd *et al.* [AAG95] vont plus loin et définissent cela explicitement en considérant que l'ensemble des conventions qui sont utilisées pour interpréter une classe de descriptions architecturales correspond à la définition d'un style.

1.1.4.2 Réutilisation de solutions

Certains styles architecturaux sont souvent décrits comme des "solutions miracle" pour toutes les formes de logiciels. Toutefois, un bon architecte doit choisir un style qui corresponde aux besoins du problème particulier à résoudre [Sha95]. Choisir le bon style architectural exige une compréhension du domaine du problème ainsi qu'une prise de conscience de la variété des styles architecturaux et des préoccupations spécifiques qu'ils adressent. Les styles architecturaux ont été considérés comme des patrons d'architecture. On peut dire qu'un patron est architectural si il se réfère à un problème du niveau architectural, c'est-à-dire si ce patron porte sur la structure globale du système et pas seulement quelques sous-systèmes [AZ05]. Cela inclut certains patrons de conception assez "gros" pour être considérés comme patrons architecturaux (*e.g.*, *Interpreter* [GHJV94]). De son coté, Garlan *et al.* proposent de considérer les styles architecturaux selon deux catégories [GAO94] :

1. Les idiomes et patrons d'architectures : cette catégorie inclut les organisations structurelles globales, telles que les systèmes en couches (*layered*), les systèmes tuyaux et filtres (*pipe&filter*), les systèmes client/serveur, les systèmes de tableau noir (*blackboard*), etc.
2. Les modèles d'architectures de référence : cette catégorie inclut des organisations qui prescrivent des configurations spécifiques de composants et de connecteurs pour des domaines d'applications bien déterminés (*e.g.*, architecture pipeline d'un compilateur). Elle inclut aussi une large gamme d'approches dédiées telles que l'avionique ou la robotique mobile.

Nous remarquons que les idiomes et patrons d'architecture sont considérés comme des abstractions fondamentales qui bénéficient d'une large portée de réutilisation, et qui au fil du temps sont intégrés dans les langages et les outils. Shaw [Sha96] ou Buschmann *et al.* [BMR*96] ont publié une liste non-exhaustive et empirique de tels patrons. Par essence, leur nombre tend à être limité et constant. En revanche, les modèles d'architectures de référence permettent de décrire de l'expertise consacrée à des développements d'applications bien précises, offrant naturellement une moindre portée de réutilisation. Ils sont à même d'exprimer des politiques spécifiques à des compagnies différentes, motivées par des préoccupations de qualité, de sécurité, de performance ou de maintenance. Par conséquent, on s'attend à ce que de larges bibliothèques de modèles d'architectures dédiés à des domaines soient créées. De plus, ces modèles forment un réservoir dynamique de connaissances. Ils évoluent plus rapidement que les idiomes et les patrons car les exigences d'un domaine d'application subissent des changements en continue, et leur procédé de validation et de publication tend à être moins rigoureux.

1.1.4.3 Relations inter-styles

Les styles peuvent être reliés les uns aux autres de différentes façons. Nous présentons ici brièvement deux des relations les plus fréquemment mentionnées dans la littérature.

Une première relation est la spécialisation. Un style peut être un sous-style d'un autre en renforçant les contraintes, ou en fournissant des versions plus spécialisées de quelques-uns des types d'éléments. Par exemple, le style *pipeline* pourrait spécialiser le style *tuyaux&filtre* en interdisant les configurations non-linéaires et en spécialisant le type d'élément "filtre" en un "palier" du pipeline qui possède un seul port d'entrée et un seul port de sortie. Autre exemple : un style client-serveur N-tiers pourrait spécialiser le style plus général client-serveur en limitant les interactions entre les tiers non-adjacents.

Une seconde relation importante est la composition, d'autant plus que la plupart des systèmes exhibent un mélange de styles dans leur composition, que Garlan et Shaw décrivent sous le terme d'architectures hétérogènes [GS93]. On peut combiner deux styles en prenant l'union de leur vocabulaire de conception, et en joignant leurs contraintes. Par exemple, on pourrait ajouter un composant de base de données à un système de type tuyau-filtre en liant un style *tuyau&filtre* avec un style base de donnée. Dans ce cas, il peut être nécessaire de définir également de nouveaux types de composants ou de connecteurs qui appartiennent à plus d'un style, comme un type de composant qui possède un comportement similaire à un filtre mais qui peut aussi accéder à une base de donnée.

Dans les travaux académiques, on remarque que la spécialisation des styles architecturaux a été traitée tôt dans [GAO94] où Garlan *et al.* ont défini une relation de sous-classement entre des styles dans leur globalité (et pas uniquement entre des types pris individuellement dans le style) ou plus récemment dans Archware ADL [Ley04]. La question de savoir quand plusieurs styles peuvent être combinés ou si ils peuvent se chevaucher dans un modèle n'est toujours pas réglée [CGL*03]. Par défaut, aucune restriction n'est placée sur cette possibilité dans les modèles de description d'architecture. C'est ainsi par exemple que Archware ADL [Ley04] permet de définir un style par agrégation de styles existants.

1.1.5 Bilan

La dernière décennie a vu une énorme augmentation de l'importance d'une branche de l'ingénierie logicielle, connue sous le nom d'architecture logicielle. "Architectes techniques" et "architecte en chef" sont des intitulés de postes qui abondent aujourd'hui dans l'industrie du logiciel. Il existe même un institut mondial des architectes logiciels [*IAW*]. Toutefois, il apparaît que le terme "architecture" est l'un des plus surexploités et des moins compris dans les milieux du développement de logiciels professionnels.

Une grande partie du travail d'un architecte vise à trouver une partition judicieuse d'une application en un ensemble de composants inter-reliés. Les différentes exigences et contraintes du projet considéré définissent le sens exact de "judicieuse" dans la phrase précédente – une architecture doit être conçue pour répondre aux exigences et aux contraintes de l'application à laquelle elle est destinée. En partitionnant l'application, l'architecte affecte des responsabilités à chaque composant. Ces responsabilités définissent les missions qu'un composant doit remplir dans l'application. De cette manière, chaque composant joue un rôle spécifique dans l'application, et l'ensemble des composants qui forment l'architecture collaborent en vue de fournir la fonctionnalité attendue. Finalement, la conception et la validation d'une architecture d'un système complexe est un exercice créatif, exigeant des connaissances, de l'expérience et de la discipline. La notion de style architectural a été introduite pour capitaliser des savoir et des savoir-faire pour la conception d'architecture. De ce point de vue, ils servent de guides aux architectes et tendent à améliorer la qualité globale des architectures produites.

1.2 Langages de description d'architectures

En accompagnement de la notion d'architecture logicielle, des formalismes sont apparus au cours des années 90 : les ADLs (*Architecture Description Languages*) ou langages de description d'architecture. Ils sont utilisés pour décrire la structure d'un système comme un assemblage d'éléments logiciels. Cela est illustré par le diagramme de la Figure 1.5 où l'on voit un client et un serveur reliés par un appel de procédure à distance (*Remote Procedure Call*).

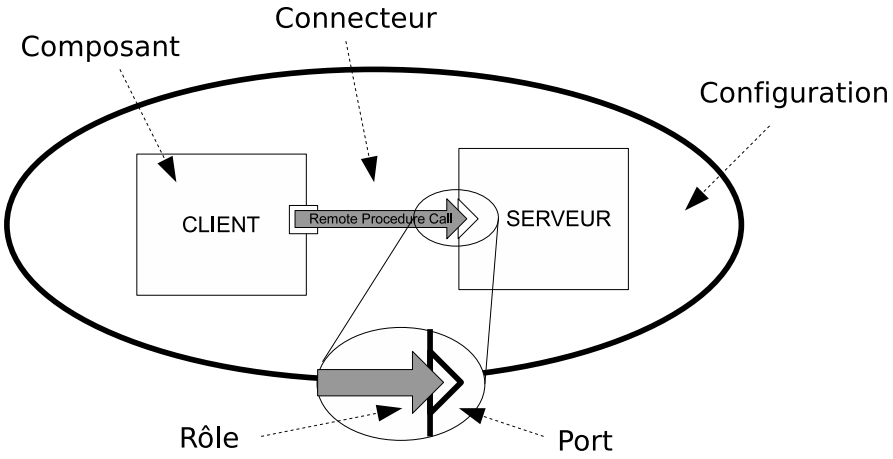

Figure 1.5 – Assemblage d'éléments architecturaux pouvant être décrit à l'aide d'un ADL.

De nombreux ADL ont été développés par le milieu académique, comme Wright, Darwin, ACME, Unicon, Rapide, Aesop, C2 SADL, MetaH, etc. [MT00]. De manière très générale, les entités manipulées par ces langages sont des éléments bien identifiés, suffisamment abstraits et les plus indépendants possible les uns des autres. La spécification d'un assemblage de tels éléments constitue leur construction fondamentale. Même si, de ce point de vue, ils affichent a priori une certaine adéquation avec la conception à base de composants lorsqu'elle vise la réalisation d'un système par assemblage de composants logiciels pré-existants (les COTS [3]), nous rappelons qu'ils ont été originellement motivés par la maîtrise de la structure de plus en plus complexe des systèmes logiciels. Dans les sections suivantes, nous listons l'ensemble des concepts que l'on trouve dans les ADLs.

1.2.1 Concepts de base

En ce qui concerne les architectures logicielles, la plupart des travaux récents publiés sont dans le domaine des langages de description d'architecture (ADL). Un ADL est, selon Taylor et Medvidovic [MT00], un langage qui offre des fonctionnalités pour la définition explicite et la modélisation de l'architecture conceptuelle d'un système logiciel, comprenant au minimum : des composants, des interfaces, des connecteurs, et des configurations architecturales. Nous donnons dans ce qui suit une définition plus précise de ces concepts minimaux.

3. *Component-Off-The-Shelf* : acronyme utilisé pour désigner des composants prêt à l'emploi.

1.2.1.1 Composant

Le *composant* représente le principal élément de calcul et de stockage des données dans un système. Un composant possède un ensemble d'interfaces, appelés ports, qui définissent les points d'interaction entre cet élément et son environnement. La taille et la complexité d'un composant est très variable. Un serveur, une base de donnée ou une fonction mathématique sont des exemples de composants. Les modèles de composants hiérarchiques acceptent le concept de *composant composite*, c'est-à-dire un composant contenant à son tour une configuration de composants et de connecteurs.

1.2.1.2 Connecteur

Le *connecteur* identifie l'interaction entre les composants. Le connecteur peut représenter une interaction simple comme une invocation à un service, ou bien un protocole complexe, comme le contrôle d'un robot sur Mars par une station de contrôle au sol. Un connecteur définit un ensemble de rôles qui permettent d'identifier les participants à l'interaction. Par exemple, un connecteur "pipe" (*e.g.*, tube de communication des systèmes Unix) possède un rôle d'écriture et un rôle de lecture ; un connecteur de type diffusion-abonnement (*i.e.*, Observateur/Observé [GAO94]) peut avoir de multiples rôles d'observateur et d'observés. Le connecteur peut disposer d'une glu qui définit comment les rôles interagissent entre eux.

1.2.1.3 Interface

Une interface constitue le "portail" des composants, des connecteurs et des configurations vers le monde extérieur. L' interface est aussi connue sous le nom de *port* pour les composants et les configurations, et de *rôle* pour les connecteurs. Par exemple, si un composant implémente une API particulière, il posséderait probablement une interface indiquant que d'autres composants peuvent faire appel à cette API sur le composant cible. Typiquement, les interfaces ont un identifiant unique, une description textuelle, et une direction (un indicateur pour savoir si l'interface est fournie, requise, ou les deux). On peut également trouver une spécification de la sémantique de l'interface. Les interactions entre les interfaces sont définies par des attachements et des bindings.

- Attachement (liaison) : représente la connexion d'un port d'un composant et un rôle d'un connecteur, si ces derniers ont des directions compatibles de type requis/fournis.
- Binding (correspondance) : les connexions entre les ports (respectivement les rôles) d'une configuration ou d'un composant composite (respectivement d'un connecteur composite) et les ports (respectivement les rôles) de leurs sous-composants (respectivement sous-connecteurs) sont appelées bindings.

1.2.1.4 Configuration

Une configuration architecturale (ou simplement architecture ou système) est un graphe qui montre la façon dont un ensemble de composants sont reliés les uns aux autres par l'intermédiaire de connecteurs. Le graphe est obtenu en associant les ports des composants avec les rôles des connecteurs adéquats, en vue de construire l'application. Par exemple, les ports des composants de type "filter" sont associés aux rôles des connecteurs de type "pipe" à travers lesquels ils lisent et écrivent des flux de données. L'analyse d'une configuration permet par exemple de déterminer

si une architecture est "trop profonde", ce qui peut affecter la performance due au trafic de messages à travers plusieurs niveaux hiérarchiques, ou "trop large", ce qui peut conduire à trop de dépendances entre les composants.

1.2.2 Concepts supplémentaires

Bien que les concepts précédents soit suffisants pour décrire une architecture logicielle, la majorité des ADLs y ont ajouté les concepts supplémentaires suivants.

1.2.2.1 Propriétés

En supplément des éléments de haut-niveau pré-cités, la plupart des architectures associent également des propriétés à leurs éléments constitutifs. Par exemple, pour une architecture dont les composants sont associés à des tâches périodiques, les propriétés pourraient définir la périodicité, la priorité, et la consommation CPU de chaque composant. Les propriétés des connecteurs pourraient inclure la latence, le débit, la fiabilité, le protocole d'interaction, etc. Les propriétés peuvent être fonctionnelles ou non-fonctionnelles.

1.2.2.2 Types

Le type est au paradigme composant ce que la classe est au paradigme objet. Il permet l'encapsulation de fonctionnalités et des éléments internes en vu d'être réutilisables. Un type peut être instancié plusieurs fois dans une même architecture ou peut être réutilisé dans d'autres architectures. Les instances des types ont les même structures et comportements que celui du type. Un système de type explicite facilite la compréhension et permet d'analyser les architectures [Gar95]. On peut ainsi réutiliser des composants, des connecteurs et des configurations par le biais de types de composants, de types de connecteurs et de types de configurations. À cet égard, un style architectural peut être vu comme un type de configuration particulier.

1.2.2.3 Contraintes

Les contraintes expriment des restrictions placées sur les composants, les connecteurs et les configurations. Elles sont spécifiées au niveau des types pour s'appliquer sur les instances. Une contrainte sur un composant peut par exemple servir à borner les valeurs de ses propriétés. Un exemple de contrainte sur un connecteur est la restriction du nombre de composants qui interagissent à travers ce dernier. Une contrainte sur une configuration peut permettre de gouverner la topologie et le nombre d'instances autorisées. Les contraintes peuvent être spécifiées soit dans un langage de contrainte séparé, soit directement en utilisant les notations de l'ADL hôte.

1.2.3 Rétrospective des ADLs

L'étude de Medvidovic et Tailor [MT00] a révélé qu'il y eu une prolifération dans les ADLs disponibles, tandis qu'il y avait beaucoup de similitudes entre eux. Néanmoins, ces ADLs ont différentes vocations, ce qui suggère la possibilité d'utiliser plusieurs ADLs pour décrire l'architecture d'un même système. Plutôt que de considérer chaque ADL, nous présentons dans la Figure 1.6 une frise chronologique des différents ADLs qui ont été proposés, en prenant soin de

mentionner leur origine. Cette rétrospective fait ressortir deux générations d'ADLs issus de trois écoles académiques.

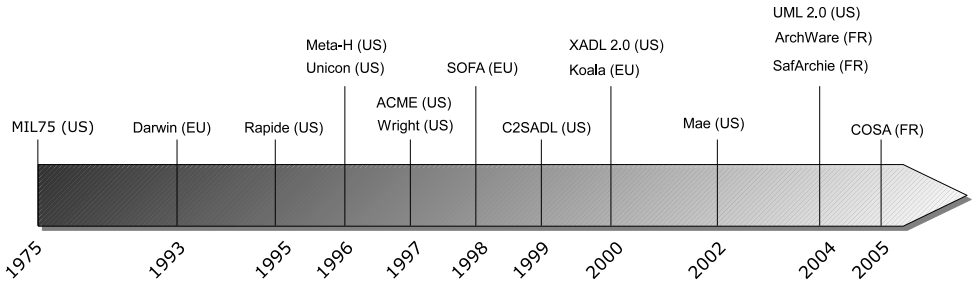

Figure 1.6 – Frise chronologique des ADLs ainsi que leur origine.

1.2.3.1 Deux générations d'ADLs

Durant la dernière décennie, une douzaine d'ADLs différents sont apparus. Chacun d'eux vise à décrire une configuration architecturale, mais selon un angle de vue différent. En outre, chacun de ces ADLs possède des règles de syntaxe très spécifiques et adaptées à la description de certaines propriétés de l'architecture. De nos jours, nous faisons référence à ces notations comme des ADLs de première génération [DI04]. Certains des exemples les plus représentatifs de cette génération sont Darwin [MK96], Wright [All97], Rapide [LV95] et C2 [MRT99].

L'étude de la première génération d'ADL a montré que, bien que différents, ces ADLs partagent une même base conceptuelle – ou une ontologie – qui détermine un ensemble de concepts et de préoccupations communes pour la description architecturale. Cela a naturellement conduit à l'idée qu'il serait possible de créer un ADL générique, qui combinerait les propriétés communes et les caractéristiques bénéfiques de la première génération d'ADL. Ces langages qui cherchent à prendre du recul sur les tentatives antérieures pour la spécification d'architectures sont considérés comme des ADLs de seconde génération [DI04]. Les exemples les plus significatifs de la deuxième génération sont ACME [GMW97] et xADL 2.0 [DHT01]. En acceptant l'idée de considérer UML 2.0 comme un ADL a part entière [ICG*04], celui-ci s'inscrit dans cette seconde génération.

1.2.3.2 Trois écoles académiques

Lorsque l'on s'intéresse aux origines des ADLs, on s'aperçoit que le paysage académique se partage entre l'école américaine (US), l'école européenne (EU) et l'école française (FR). Historiquement, MIL75 [DK75] – considéré comme l'ancêtre des ADLs– fût avancé par les américains, dont les travaux sont restés très influents jusqu'à aujourd'hui. La contribution de l'école américaine est issue essentiellement de la CMU (*Carnegie Mellon University*) et l'UCI (*University of California, Irvine*), dont les travaux sont respectivement pilotés par David Garlan *et al.* et Richard N. Taylor *et al.*. Dans le même temps, UML 2.0 est un poids lourd du domaine qui est formalisé par l'OMG (*Object Management Group*), un consortium américain ouvert et sans but lucratif (> 800 membres). De son coté, l'école européenne s'est illustrée très tôt avec la contribution de l'Imperial College avec Darwin, puis plus tard Koala [vOvdLKM00]. Moins connus, Conic [KM85] et BackBone [MKM06] sont également deux ADLs issus de cette entité anglo-saxone. Quant à SOFA [PBJ98], c'est un ADL développé par la Charles University, située en République

Tchèque. Une école française a commencé à émerger ces dernières années, avec SafArchie [BD04], Archware ADL [MKB*04] et enfin COSA [Sme06, KSO05], un ADL proposé par notre équipe. En dehors du cercle académique, on notera tout de même la participation des industriels aux ADLs Meta-H [BEJV96] (Honeywell Inc.) et Koala (Philips).

1.2.4 Bilan

Jusqu'à présent, un certain nombre d'ADLs ont été développés afin d'aider le développement centré architecture. Les ADLs fournissent des notations formelles pour décrire et analyser les systèmes logiciels. Ils sont généralement accompagnés par divers outils destinés à analyser, simuler, et parfois générer le code des systèmes modélisés. Un certain nombre d'ADL offre également un support pour modéliser le comportement et les contraintes sur les propriétés des composants et des connecteurs [MT00]. Ces comportements et contraintes peuvent être exploités pour assurer la cohérence d'une configuration architecturale tout au long de la vie d'un système (*e.g.*, en démontrant la conformité entre les services de composants interagissants).

Le consensus sur les concepts fondamentaux que doit offrir un langage de description d'architecture a été long à atteindre. Les ADLs de seconde génération sont nés sur la base des efforts de standardisation et d'interopérabilité entrepris auparavant. A ce titre, ACME a été un précurseur en se positionnant dès le départ comme un ADL pivot et susceptible de fédérer l'ensemble des travaux de la communauté. Par ailleurs, l'émergence tardive de l'école française fait qu'elle bénéficie d'un recul supplémentaire qui l'a orienté vers des problématiques traitées de façon limitée comme l'évolution par exemple. Enfin, il faut bien dire que l'approche ADL issue des milieux académiques ne s'est pas imposée dans le monde industriel. Il lui est souvent reprochée d'utiliser des notations difficiles à mettre en œuvre et de fournir des outils peu exploitables.

Dans la section suivante, nous nous intéressons aux niveaux de modélisation qui existent lorsque l'on utilise les ADLs. Cette vision de l'architecture logicielle est une particularité de notre équipe, et sera importante à prendre en compte pour la suite de notre travail sur l'évolution architecturale.

1.3 Architecture et méta-modélisation

A travers notre étude des ADLs, nous constatons que la spécification d'une architecture peut passer par plusieurs niveaux de modélisation. Dans cette section, nous appliquons les quatre niveaux de modélisation proposés par l'OMG à l'architecture logicielle. En effet, la technique de méta-modélisation peut reposer sur une approche objet (comme celle de l'OMG), ou bien sur une approche composant. On parlera ainsi respectivement de méta-modélisation par objet et de méta-modélisation par composant. Nous nous plaçons dans le second cas de figure et le résultat de cela est une hiérarchie à différents niveaux d'architecture, démarrant sur la définition d'une méta-méta-architecture jusqu'à une application.

1.3.1 Principes de la méta-modélisation

Avant d'étudier ce qu'est un méta-modèle, il semble logique de s'intéresser d'abord à ce qu'est un modèle. Minksy propose la définition suivante [Min68] : un objet A* est un modèle d'un objet A pour un observateur O dans la mesure où O peut utiliser A* pour répondre aux questions qu'il

se pose sur A. Dans ce cas, O utilisera le modèle A* comme support de raisonnement. L'action de modélisation permet donc de passer de l'objet du modèle (A dans la définition de Minsky) vers le modèle (A*), à travers un principe d'abstraction visant à simplifier la représentation de l'objet à modéliser. Il est possible de poursuivre ce travail d'abstraction pour créer des méta-modèles. La création d'un méta-modèle revient à construire une représentation d'un modèle, toujours dans le but de fournir un support de raisonnement. L'action de méta-modélisation permet donc de passer du modèle (A* dans la définition de Minsky) vers le méta-modèle (A**).

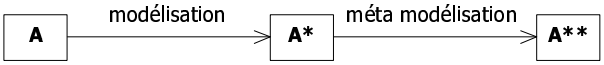

Figure 1.7 – Activité de modélisation et de méta-modélisation dans les termes de Minsky.

Comme le résume la Figure 1.7, l'objet A** ne peut pas être obtenu directement par modélisation de l'objet A, mais bien par la modélisation de l'objet A*. Cette relation d'ordre dans l'activité de modélisation fait apparaître des niveaux de modélisations distincts.

1.3.2 Méta-modélisation par composant

La principale notation pour la méta-modélisation par objet est le MOF (*Meta Object Facility*) [Obj02]. L'OMG a défini deux variantes de MOF : EMOF (MOF essentiel) et CMOF (MOF complet). Le but du MOF est de définir un langage unique et standard pour décrire des méta-modèles. Il est constitué d'un ensemble relativement restreint (bien que non minimal) de concepts "objets" permettant de modéliser ce type d'information. Par exemple, UML est l'un des méta-modèles décris en utilisant le MOF [4]. D'autres notations pour la méta-modélisation objet existent. Parmi elles, on peut citer KM3 [JB06], ECORE [BMS08] ou Kermeta [MFJ05].

Très peu de travaux ont été conduits dans la technique de méta-modélisation composant. AML (*Architecture Meta-Language*) [Wil99] est une première tentative de proposition d'un socle pour fournir aux ADLs une solide base sémantique. MADL (*Meta Architectural description Language*) [SOK05] est une seconde tentative proposée par notre équipe. MADL est l'équivalent du MOF pour l'architecture logicielle. MADL inclut les concepts et les mécanismes des langages de description d'architecture, et conserve l'instanciation, l'héritage et la composition, issues de l'orientation objet.

	Modélisation par objet	Modélisation par composant
3 – Niveau méta-méta-modèle	MOF	MADL
2 – Niveau méta-modèle	UML, CWM, SPEM, etc.	ACME, COSA, C2, etc.
1 – Niveau modèle	Modèles	Architectures
0 – Niveau instance	Informations	Applications

Table 1.1 – Hiérarchie selon une méta-modélisation par objet et par composant.

En pratique, un méta-méta-modèle détermine le paradigme utilisé dans les modèles qui sont construits à partir de lui. De ce fait, le choix d'un méta-méta-modèle est un choix important où la question de l'objet ou du composant comme unité de construction de base est posée.

4. UML 2.0 est parfois considéré – à tord – comme un langage de méta-modélisation car son diagramme de classes est proche du MOF.

En choisissant MADL, nous optons pour le composant et nous identifions une hiérarchie de modèles architecturaux, comme le montre la table 1.1. Les quatre niveaux de modélisation d'une architecture sont détaillés ci-après.

1.3.3 Niveaux de modélisation d'une architecture

Nous identifions quatre niveaux de modélisation dans les systèmes : le niveau méta-méta-architecture, le niveau méta-architecture, le niveau architecture et le niveau application. Nous schématisons ces niveaux sous forme d'une "architecture égyptienne" présentée dans la Figure 1.8, où chaque étage de la pyramide se conforme à l'étage immédiatement supérieur.

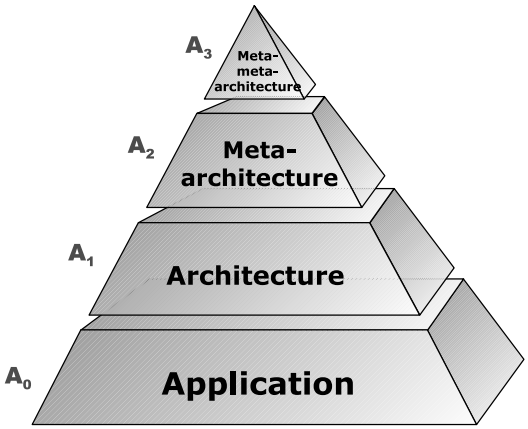

Figure 1.8 – Les quatre niveaux d'architecture que nous proposons.

Le niveau méta-méta-architecture (A3) : fournit les éléments minimaux de modélisation d'une architecture. Une méta-méta-architecture se conforme à elle-même (*i.e.*, s'auto définie). Les concepts de base d'un méta ADL sont représentés à ce niveau.

Le niveau méta-architecture (A2) : fournit les éléments de modélisation de base pour un langage de description d'architecture (ADL) : composant, connecteur, architecture, ports, rôles, etc. Ces concepts de base permettent de définir différentes architectures. Les méta-architectures se conforment aux méta-méta-architectures. Dans le cadre d'une relation de conformité, chaque élément de A2 est associé à un élément de A3.

Le niveau architecture (A1) : à ce niveau, plusieurs types de composants, de connecteurs et d'architectures sont décrits. Les architectures se conforment aux méta-architectures (ADLs), donc chaque élément de A1 est associé à un élément de A2.

Le niveau application (A0) : est le niveau où les unités d'exécution sont localisées. Une application est vue comme un ensemble de composants, de connecteurs et de configurations. Les applications sont conformes au niveau architecture. Chaque élément de A0 est associé à un élément de A1.

1.3.4 Exemple illustratif

Pour donner une idée de l'utilisation des différents niveaux de modélisation, prenons l'exemple d'un système de type client/serveur. D'une part la Figure 1.9 ne fais pas apparaître le niveau A3 (ce dernier présentant peu d'intérêt pour notre travail) et d'autre part, la relation de conformité entre deux niveaux de modélisation est réalisée par un principe d'instanciation. Pour cet exemple, nous nous basons sur les concepts de l'ADL ACME, qui réifie l'ensemble des éléments architecturaux évoqués dans ce chapitre et supporte une modélisation explicite des styles architecturaux – appelés familles. La notation graphique est celle du diagramme de composant d'UML 2.0. Il est à noter qu'avec cette notation le connecteur n'existe pas en tant que tel, mais est obtenu par un assemblage des concepts UML suivants : une interface et deux relations qui sont la réalisation et l'usage.

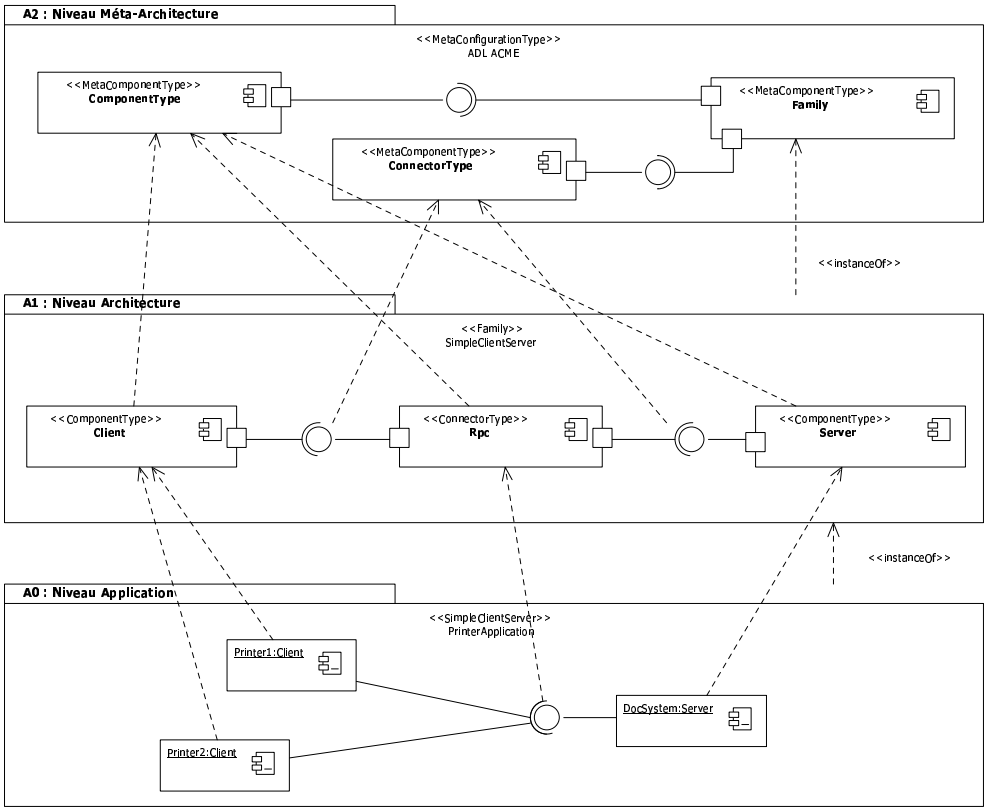

Figure 1.9 – Illustration des niveaux de modélisation de l'architecture d'un système de type client/serveur en ACME.

Le niveau méta-architecture (A2) : c'est à ce niveau que les éléments de modélisation de l'ADL ACME sont décrits. En l'occurrence, ACME permet de décrire des styles architecturaux appelés familles (`Family`) à travers la spécification de types de composants (`ComponentType`) et de types de connecteurs (`ConnectorType`) pour les relier.

Le niveau architecture (A1) : le style architectural client/serveur dans sa version simplifiée est spécifiée en ACME par la création des types de composants `client` et `server` et du type de connecteur `RPC` (*Remote Procedure Call*).

Le niveau application (A0) : c'est là que l'on trouve l'application proprement dite, constituée des deux composants clients `Printer1` et `Printer2`, d'un composant serveur `DocSystem` et des deux connecteurs `RPC` nécessaires à leurs communications.

1.3.5 Bilan

On peut citer les ADLs qui distinguent les types des instances, dans lesquels chaque élément (*i.e.*, interface, composant, connecteur) doit être défini avant d'être instancié, et les ADLs qui permettent de décrire directement ces éléments. Selon le cas, l'architecture logicielle est décrite sur un ou deux niveaux de modélisation. De manière générale, l'acte de (méta)modélisation n'a pas imprégné les travaux de notre communauté où le terme "architecture" est surchargé, désignant tantôt le niveau A1, tantôt le niveau A0. L'avantage de combiner une technique de méta-modélisation avec le domaine de l'architecture logicielle est de :

- aider à la compréhension de la description de l'architecture logicielle ;
- identifier des concepts réifiés de ceux qui ne le sont pas ;
- distinguer des liens entre les niveaux de modélisation.

D'une part, cette vision de l'architecture logicielle est importante lorsque la problématique de l'évolution sera abordée dans le chapitre suivant. D'autre part, disposer d'un référentiel commun comme celui schématisé par "l'architecture égyptienne" de la Figure 1.8 permet de communiquer sans ambiguïtés autour des architectures logicielles et de disposer d'un vocabulaire partagé. Par exemple, nous considérons que le terme de composant réfère au niveau A0, tandis que le terme de type de composant réfère au niveau A1. Ceci rejoint la distinction nette entre l'objet et sa classe dans la terminologie du paradigme objet.

1.4 Conclusion

Nous avons présenté dans ce chapitre le contexte global dans lequel s'inscrit cette thèse : le domaine des architectures logicielles à base de composants. Nous avons montré comment l'architecture logicielle s'est progressivement retrouvée sur le devant de la scène, et quelles nouvelles pratiques d'ingénierie sont apparues avec elle. Pour mieux comprendre ce que recouvre l'architecture logicielle, nous avons synthétisé les définitions notables proposées dans la littérature. Nous avons également consacré une section à la notion de style architectural, que nous considérons comme un ingrédient clé du développement centré architecture. Puis, nous avons abordé les travaux sur les langages de description d'architectures, sur la base de leurs concepts communs. Nous avons clôturé ce chapitre par la description de l'architecture logicielle à travers ses différents niveaux de modélisation.

Les leçons générales que nous tirons de ce vaste ensemble de travaux sur les architectures logicielles est que l'accroissement de la complexité impose de monter en abstraction afin de garder le contrôle, pouvoir comprendre et communiquer autour d'un système. Par dessus tout, la capitalisation et la réutilisation des expériences passées via des solutions éprouvées est une

technique phare pour guider et améliorer les activités des architectes. Nous sommes convaincus que ces grands principes doivent être portés au domaine de l'évolution dans les architectures à base de composants. En outre, nous sommes certains de la nécessité de prendre appui sur la méta-modélisation pour traiter correctement la problématique de l'évolution architecturale. Cette problématique fait l'objet du chapitre suivant.

Chapitre 2

Évolution architecturale : analyse et synthèse

Après avoir présenté dans le chapitre précédent les définitions des notions et concepts relatifs aux architectures logicielles, à leurs langages de description ainsi qu'a leur méta-modélisation, nous abordons dans ce chapitre la problématique de l'évolution structurelle des architectures logicielles à base de composants. Aborder la problématique de l'évolution au niveau de l'architecture permet à l'architecte de raisonner à un haut niveau d'abstraction sur l'ajout, la suppression, la modification des composants et/ou des connecteurs ou encore la réorganisation de l'architecture, sans pour autant se noyer dans les détails de l'implémentation de ces éléments. Clairement, les concepts et les mécanismes offerts à un architecte pour *spécifier* et *gérer* l'évolution de son architecture dépendent du *modèle d'évolution* considéré.

Un point de départ évident de cette recherche consiste à étudier les modèles d'évolution qui ont été proposés dans la littérature. Dans ce but, nous commençons par construire un cadre de comparaison pour pouvoir caractériser et comparer les différents modèles d'évolution de manière indépendante les uns des autres. Ce cadre de comparaison prend la forme d'une pyramide des besoins, en proposant une hiérarchisation de ce que peut offrir un modèle d'évolution. A chaque niveau de la pyramide des besoins sont associés des critères afin d'exhiber un certains nombre d'aspects que nous jugeons importants concernant l'évolution. Dans certains cas, ces critères sont raffinés en sous-critères. Puis, nous proposons une analyse des principaux modèles d'évolution que l'on trouve dans les travaux sur les ADLs de l'école américaine, européenne et française. Nous nous appuyons sur cette analyse et sur la pyramide des besoins pour établir un bilan récapitulatif. Ce dernier souligne à la fois les limites des modèles d'évolutions étudiés et permet de poser les fondations d'un nouveau modèle d'évolution qui s'inscrira comme le cœur de ce travail de thèse.

2.1 Évolution architecturale

L'évolution logicielle a tendance à être progressive et incrémentale, dirigée, par exemple, par les retours des utilisateurs ou d'autres intervenants, sous la forme de rapports d'erreurs et de demandes de nouvelles fonctionnalités, ou plus généralement, par les changements des exigences fonctionnelles ou non-fonctionnelles. La plupart des systèmes logiciels sont donc exposés à de nombreuses forces qui exigent leur modification. En conséquence, le logiciel change continuellement. Ce phénomène inévitable est appelé *évolution logicielle* [LB85], et a été souligné par

les travaux de Lehman *et al.* dès les années 70. En pratique, il n'y a pas de définition précise et largement acceptée de ce que doit être l'évolution logicielle, selon le niveau d'abstraction et le niveau de granularité considéré. Cependant, nous pouvons formuler trois observations majeures. Premièrement, l'évolution peut avoir lieu durant le développement ou après la mise en exploitation du logiciel. Ce dernier cas est souvent assimilé à une activité de maintenance. Deuxièmement, l'évolution se manifeste par des modifications, très simples ou très complexes, jusqu'à la restructuration totale du logiciel. Troisièmement, l'évolution peut concerner les différents artefacts produits lors du cycle de développement du logiciel : code source, modèle de conception, architecture, documentation, etc. Lorsque l'évolution concerne spécifiquement l'architecture des systèmes, on parle d'*évolution architecturale* – ou encore d'évolution basée architecture. Dans de nombreux systèmes, l'évolution architecturale peut fournir la capacité de modifier l'ensemble des fonctionnalités offertes par le système, dans un effort de personnalisation et d'extensibilité attendue par les utilisateurs finaux. Dans ce qui suit, nous explorons dans quelle mesure l'architecture logicielle offre une fondation saine pour aborder la problématique de l'évolution.

2.1.1 Motivation

Il existe différentes stratégies pour faire face à l'évolution. L'approche classique en ré-ingénierie consiste à faire évoluer les logiciels au niveau de leur code. Une approche plus récente consiste à faire évoluer les logiciels au niveau de leur modèle [1]. Dès lors qu'un système dispose d'une architecture explicite, c'est-à-dire d'un modèle de son architecture, il est envisageable d'aborder la problématique de son évolution sur ce dernier. Nous qualifions d'évolution à grande échelle (*evolving-in-the-large*) l'approche qui consiste à faire évoluer un système au niveau de son architecture plutôt qu'au niveau de son code source. A l'instar du glissement de paradigme qui s'est opéré pour le développement des systèmes complexes (cf. chapitre 1), le niveau architectural offre un niveau d'abstraction et de raisonnement bien supérieur aux lignes de code, permettant à l'architecte de décider et de réaliser des évolutions d'ensemble sur son système. En outre, il est admis que le niveau architectural joue un rôle bénéfique dans l'évolution en exposant les dimensions (complexité, coûts, etc.) selon lesquelles un système peut évoluer [Gar00a]. Ainsi, l'importance de l'architecture logicielle pour l'évolution des systèmes devient de plus en plus évident.

2.1.2 Pour quels usages ?

Considérons un scénario dans lequel une société développe un logiciel innovant. Conformément aux bonnes pratiques du génie logiciel, la société développe en premier lieu une bonne architecture pour son logiciel dans un style architectural approprié, puis modélise son architecture avec un ADL, raffine ensuite l'architecture en une conception détaillée (*e.g.*, via une méthodologie objet) et enfin implémente l'application (*e.g.*, via une technologie objet). Le nouveau logiciel connaît un succès immédiat et de nombreux exemplaires sont vendus. Motivé par ce succès, la société entame un cycle d'avancements rapides, créant des "add-ons", vendant des mises à jour, adaptant le logiciel à différentes plateformes ou encore personnalisant l'application pour différents clients. Dans ce cas de figure, l'architecture évolue pour refléter l'évolution du système logiciel qu'il est censé représenter [MR97]. Une perte de synchronisation entre un sys-

1. Dans un certain sens, l'évolution de modèle englobe l'évolution de code, au moins pour ceux qui considèrent qu'un programme n'est rien d'autre qu'un type spécial (très détaillé) de modèle.

tème et son architecture – appelé phénomène d'érosion architecturale [PW92] – remet en cause la valeur de l'architecture puisque cette dernière ne fourni plus une image fidèle du système.

Par ailleurs, les activités de maintenance logicielle ont été déplacées du niveau du code source vers le niveau de l'architecture. Le mise à jour d'un composant pour utiliser une toute nouvelle librairie par exemple, peut être réalisée en changeant ses connexions architecturales de façon à ce qu'il soit lié à cette nouvelle librairie plutôt que de changer le code source du composant. A travers le niveau d'abstraction élevé qu'ils offrent, les modèles architecturaux sont plus faciles à comprendre que les artefacts du niveau inférieur et fournissent une "vue d'ensemble", essentielle dans la réussite d'une maintenance. En règle générale, travailler à ce niveau d'abstraction est intéressant si les changements impactent l'architecture du système, comme c'est souvent le cas avec de la maintenance perfective (qui représente près de 65% des coûts de maintenance [Som95]), mais plus rarement avec de la maintenance corrective ou adaptative.

2.1.3 Avantages et défis

Les architectures logicielles à base de composants offrent des défis intéressants dans l'étude de l'évolution. Oreizy *et al.* [OT98] par exemple, mettent en avant plusieurs avantages résultant directement de la gestion de l'évolution au niveau de l'architecture :

1. Le contrôle de la portée et de la politique des évolutions est placée entre les mains des architectes, où les décisions peuvent ainsi être prises sur la base d'une véritable compréhension des exigences et de la sémantique de l'application.
2. Les architectes utilisent communément l'architecture logicielle pour décrire, comprendre et raisonner de façon globale sur un système. Valoriser la connaissance de l'architecte à ce niveau d'abstraction tend à faciliter la gestion de l'évolution.
3. Si aucune restriction particulière n'est placée sur la spécification interne des composants amenés à évoluer, alors il devient possible d'accueillir des composants sur étagères (COTS).
4. Les connecteurs facilitent la séparation du comportement spécifique à l'application des décisions relatives à la portée et la politique des évolutions, permettant à ces derniers d'être modifiés indépendamments.

Un autre point d'intérêt pour l'évolution est le support de familles d'architectures partageant des caractéristiques communes. Une façon de supporter les familles dans les ADLs est de séparer les types de composants et de connecteurs de leurs instances. Intuitivement, toutes les architectures issues de la même famille évoluent selon les mêmes stratégies ou "patterns". D'autre part, il est à noter que certaines familles d'applications sont amenées à évoluer plus facilement que d'autres [Gom05]. Récemment, on observe que l'évolutivité est considéré comme un attribut de qualité à part entière des styles architecturaux [OMT08]. Des investigations en ce sens pourraient par exemple aboutir sur une meilleure compréhension des limitations placées sur les types d'évolutions acceptables par un style architectural donné.

2.2 Cadre de comparaison des modèles d'évolution

Une comparaison dédiée à l'évolution dans les architectures logicielles est nécessaire afin de bénéficier d'une compréhension plus profonde de l'évolution. Plusieurs classifications et évaluations ont été proposées dans le domaine de l'architecture logicielle [SG95, Cle96, MT00] mais le

travail effectué ne permet pas de comparer les architectures évolutives car ils contiennent très peu de critères relatifs à l'évolution. En fait, seul le travail de Medvidovic *et al.* [MT00] tient compte de l'évolution dans son évaluation. Si l'on considère une taxonomie basée sur l'objectif de l'évolution, on peut citer les travaux de Lientz et Swanson [LS80], où ces derniers font la distinction entre la maintenance perfective, adaptative, corrective et préventive. Également, dans [CHK*01], on trouve une taxonomie de la maintenance et de l'évolution divisée en 12 catégories distinctes. Malheureusement, ces taxonomies dans le domaine de l'évolution sont trop générales et manquent d'informations importantes au regard des problématiques qui relèvent du niveau architectural des systèmes. Afin de répondre au besoin de comparaison ciblé, nous proposons un ensemble de critères basés sur les informations que nous désirons comprendre concernant l'évolution.

2.2.1 Pyramide des besoins

Afin de disposer d'un outil d'évaluation, nous définissons une pyramide des besoins. La pyramide des besoins schématise une hiérarchisation dans les besoins traités par un modèle d'évolution. La pyramide est constituée de trois niveaux principaux (cf. figure 2.1). Un modèle d'évolution cherche à satisfaire les besoins d'un niveau donné avant de s'attaquer aux besoins situés au niveau immédiatement supérieur de la pyramide. Ainsi, à la base de la pyramide, on trouve l'*intention* du modèle d'évolution, c'est-à-dire les objectifs que ce dernier est en mesure d'adresser. Au niveau immédiatement supérieur on trouve l'*expressivité* du modèle d'évolution. Les besoins en terme d'expressivité déterminent sa richesse sémantique. Enfin, le sommet de la pyramide concerne la *qualité* du modèle d'évolution, représentant la valeur ajoutée du modèle d'évolution. Les besoins appartenant aux différents niveaux sont expliqués en détail dans les sections suivantes.

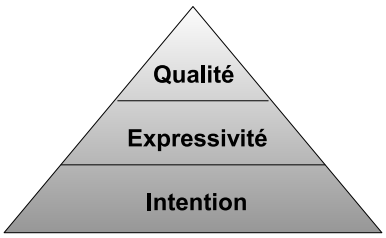

Figure 2.1 – Pyramide des besoins traités par un modèle d'évolution.

Il importe de remarquer que cette pyramide peut être exploitée pour évaluer soit une problématique d'évolution, soit un modèle d'évolution. Dans le premier cas, la pyramide sert à spécifier des desideratas, en précisant pour chaque niveau les besoins que l'on souhaite associer à l'évolution architecturale. Dans le second cas, elle sert à positionner et à comparer les modèles d'évolutions. Pour ce travail d'analyse et de synthèse, nous nous situons dans le second cas.

2.2.2 Intention d'un modèle d'évolution

Le premier niveau de la pyramide des besoins détermine les objectifs possiblement traités par un modèle d'évolution. Selon Oussalah [Ous99], ces objectifs concernent la capacité à (i) formuler l'évolution, (ii) gérer les impacts de l'évolution et (ii) garder la trace entre le modèle de

départ (avant évolution) et le modèle d'arrivé (après évolution). Dans ce qui suit, nous donnons les définitions théoriques que nous associons à ces besoins.

2.2.2.1 Formuler l'évolution

Notre vision est que toute évolution dans une architecture est causée par des opérateurs d'évolution qui peuvent être appliqués sur des éléments architecturaux, que nous qualifions alors d'éléments évolutifs. En outre, nous faisons une distinction entre les opérateurs simples et les opérateurs complexes. Les opérateurs complexes sont caractérisés par le fait qu'ils sont obtenus à partir d'opérateurs plus simples. Une opération d'évolution correspond à la combinaison d'un opérateur et d'un ensemble d'éléments évolutifs nécessaires à son fonctionnement, qui jouent alors le rôle d'opérandes. Cette vision est illustrée à travers la Figure 2.2.

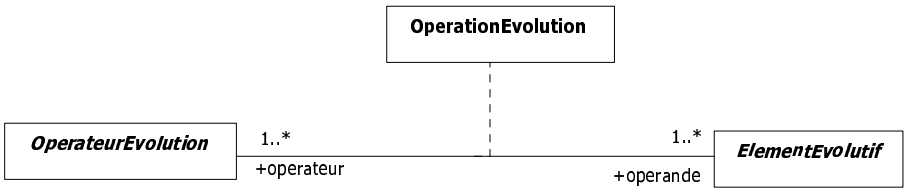

Figure 2.2 – Trois concepts pour formuler l'évolution : Opération, Opérateur et Élement évolutif.

Nous avons répertorié dans le tableau 2.1 les types d'opérations d'évolutions récurrentes dans les ADLs pour formuler les changements dans une architecture logicielle. Ces types d'opérations sont volontairement décrits à un haut niveau d'abstraction, afin de rester indépendants d'un ADL en particulier. On retrouve dans la partie supérieur du tableau les opérations que nous considérons comme simples, et dans la partie inférieure, les opérations considérées comme complexes.

TYPE D'OPERATION	DESCRIPTION
`Add` <elem_type><elem_name1[=value]> `to` <elem_name2>	Introduit un élément architectural de type <elem_type> nommé <elem_name1> (affecte optionnellement sa valeur) à un autre élément architectural <elem_name2>
`Modify` <elem_name1> `to` <elem_name2|value>	Change la sémantique d'un élément architectural en modifiant son nom de <elem_name1> vers <elem_name2>, ou en affectant une nouvelle valeur à <elem_name1>
`Remove` <elem_type><elem_name1> `from` <elem_name2>	Retire un élément architectural de type <elem_type> et de nom <elem_name1> d'un autre élément architectural <elem_name2>
`Replace` <elem_type><elem_name1> `by` <elem_name2>	Substitue un élément <elem_name2> en lieu et place d'un élement <elem_name1> de même type <elem_type>
`Connect` <elem_name1> `to` <elem_name2>	Définit une relation entre les éléments architecturaux <elem_name1> et <elem_name2>
`Disconnect` <elem_name1> `from` <elem_name2>	Supprime une relation entre les éléments architecturaux <elem_name1> et <elem_name2>
`Split` <elem_type><elem_name> `into` <elem_name_list>	Sépare un élément architectural de type <elem_type> et de nom <elem_name> en deux (ou plus) éléments définis dans <elem_name_list>
`Unify` <elem_type><elem_name_list> `into` <elem_name>	Groupe deux (ou plus) éléments définis dans <elem_name_list> dans l'élément architectural <elem_name>
`Version` <elem_type><elem_name>	Dérive une nouvelle version de l'élément architectural de type <elem_type> et de nom <elem_name>

Table 2.1 – Types d'opérations d'évolution récurrentes et leurs descriptions.

2.2.2.2 Gérer les impacts de l'évolution

Un impact représente l'effet d'un élément évoluant sur un autre. L'impact peut être pensé comme la conséquence d'un changement. Ainsi, une évolution locale peut être reconsidérée comme une évolution globale, par l'intermédiaire d'un mécanisme de propagation d'impact. De ce point de vue, l'évolution est un processus à "effet de bord", c'est-à-dire que la modification d'un élément entraîne bien souvent de nombreuses autres modifications sur d'autres éléments. Ainsi, identifier tous les impacts d'une évolution revient à s'intéresser à la portée de celle-ci. La capacité de l'architecte à déterminer la portée d'une évolution aide à raisonner sur cette dernière.

L'impact provoqué par une évolution peut varier en importance et en type selon l'élément évolutif concerné. Un impact peut être évalué à travers ses répercussions sur un ou plusieurs aspects architecturaux. Les aspects peuvent être structuraux (*e.g.*, composants, connecteurs, et la topologie tel que le style architectural utilisé, etc.), qualitatifs (*e.g.*, les attributs de qualité tels que la performance ou la maintenabilité), ou bien encore économique (les incidences en termes de coûts). Basson *et al.* [Bas98] ont proposé une typologie incluant les dimensions principales de tout impact :

- les impacts structurels,
- les impacts logiques,
- les impacts fonctionnels,
- les impacts qualitatifs,
- les impacts comportementaux.

Dans cette thèse, nous nous intéressons exclusivement à l'évolution structurelle et donc aux impacts de type structurel. Les impacts structurels correspondent aux variations structurelles de l'ensemble des éléments évolutifs. L'évolution structurelle est définie comme une modification de la structure d'un élément évolutif. Par exemple, la variation structurelle d'une configuration correspond à l'ajout ou le retrait de composants ou de connecteurs, etc. Une gestion efficace des impacts empêche l'introduction d'incohérences structurelles dans l'architecture. Par exemple, en l'absence d'un mécanisme de propagation d'impacts, la suppression d'un connecteur laissera des attachements "flottants" dans l'architecture : elle n'est plus cohérente. A l'inverse, si les attachements sont supprimés par propagation, alors l'architecture demeure cohérente du point de vue structurel.

2.2.2.3 Garder la trace de l'évolution

Cette troisième intention cherche à garder trace de l'architecture ayant évoluée en conservant un enregistrement des modifications architecturales tout au long de la durée de vie du système. On parle parfois dans ce cas d'évolution continue, par opposition à de l'évolution avec rupture [TO03]. Sont enregistrées aussi bien les évolutions atomiques que les évolutions transactionnelles ; par exemple, une évolution atomique serait l'addition d'un unique composant, tandis qu'une évolution transactionnelle pourrait inclure l'addition et la suppression de multiple composants et connecteurs.

Ainsi, on peut voir les évolutions d'une architecture sous forme d'un graphe orienté indiquant les configurations architecturales connectées par les évolutions qui les ont engendré (cf. figure 2.3). Un tel graphe peut être défini comme un couple $G = (N, E)$, où N est un ensemble de noeuds représentant des configurations architecturales et E est un ensemble d'arcs unidirectionnels connectant les noeuds tel que $E = \{(x, y)|x, y \in N\}$. Les cycles – c'est-à-dire les chemins

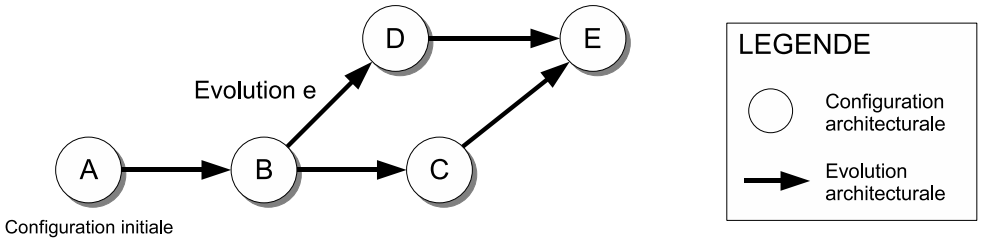

Figure 2.3 – Représentation schématique des traces des évolutions dans une architecture.

commençant et terminant sur un même noeud – dans G sont admis afin de représenter les évolutions qui font passer un système dans une certaine configuration existante, mais les boucles – c'est-à-dire les arcs ayant le même noeud source et cible – ne sont pas admis puisqu'ils représentant des évolutions qui n'ont débouché sur aucun changement de configuration. Un seul et unique arc est admis entre les noeuds, car des évolutions architecturales distinctes ne donnent pas lieu à des configurations identiques, tandis que les arcs "inverses" – c'est-à-dire des arcs connectant les même noeuds mais avec des directions opposées – sont admis afin de représenter des évolutions qui ramènent le système à sa précédente configuration.

2.2.3 Expressivité d'un modèle d'évolution

Le second niveau de la pyramide des besoins traite du pouvoir expressif d'un modèle d'évolution. Nous avons identifiés cinq besoins impliqués dans le pouvoir expressif : le niveau d'abstraction, le mode d'expression, le domaine, le niveau de modélisation et la mécanique opératoire.

2.2.3.1 Niveau d'abstraction

Le niveau d'abstraction permet d'exprimer les degrés de raffinement d'une évolution, et ce depuis sa spécification, considéré comme le plus haut niveau d'abstraction, à son implémentation représentant le plus bas niveau. Un raffinement complet permet ainsi de refléter les évolutions définies au niveau architectural vers le code source [2]. A chaque niveau d'abstraction peut lui être associé un niveau de transparence qui définit le niveau de visibilité des détails internes d'une évolution lors de sa réutilisation. Le niveau de transparence d'une évolution peut être de type :

- Boîte noire : l'évolution est réutilisée sans se soucier de son fonctionnement interne. Dans ce cas, il est impossible de la modifier selon ses besoins. Cependant, la possibilité de remplacer une évolution par une autre évolution qui poursuit le même objectif est offerte.
- Boîte grise : il s'agit d'un niveau de transparence intermédiaire. Les détails du fonctionnement d'une évolution peuvent être révélés pour sa compréhension, mais ne peuvent être sujets à des modifications émanant de l'extérieur.
- Boîte blanche : l'évolution rend transparent tous les détails de son fonctionnement interne. Ce type d'évolution a l'avantage de fournir toute l'information relative à sa mise en œuvre, et peut être modifiée selon les besoins.

2. Notons que, en ce sens, la génération de code est simplement un cas spécial de raffinement.

2.2.3.2 Mode d'expression

Le mode d'expression permet de décrire les différents modèles de représentation d'une évolution (représentation textuelle, graphique, logique, code source, ...). La description d'une évolution doit être simple, compréhensible selon une sémantique claire, mais pas nécessairement définie de manière formelle. En effet, les évolutions peuvent requérir différents modes d'expression pour des usages différents. Par exemple, une représentation graphique de haut-niveau d'une évolution peut être utilisée comme un moyen de compréhension et de communication ; une autre représentation formelle peut être utilisée pour analyser les interactions entre les évolutions ; et encore une autre représentation liée à une plateforme technologique cible pour pouvoir y exécuter les évolutions. Le mode d'expression d'une évolution dépend également de sa granularité. La notion de granularité d'évolution recouvre, selon les langages de représentation, aussi bien des opérations atomiques que de véritables opérations complexes. Du point de vue de la granularité, nous pouvons citer :

- les évolutions dites simples ou de fine granularité : la granularité d'une évolution est liée à une construction syntaxique de l'ADL, comme les primitives d'ajout, de suppression, d'attachement et de détachement. Dans ce cas, l'assemblage des évolutions est à la charge de l'architecte.
- les évolutions dites complexes ou à forte granularité : l'évolution s'apparente à une véritable tâche complexe, obtenue par assemblage d'évolutions existantes, elles-mêmes simples ou complexes. Dans ce cas, l'architecte réutilise des évolutions complètes, réduisant ainsi son travail d'assemblage.

2.2.3.3 Domaine

Le domaine reflète les différentes évolutions structurées en couches. Cette structuration permet à une évolution appartenant à une couche particulière de ne pouvoir interagir qu'avec seulement les évolutions de la même couche ou de la couche adjacente. Les évolutions du "tout domaine" constituent la couche de plus bas niveau et sont indépendant du domaine d'application. Au niveau plus haut (domaine connexe), on trouve des évolutions appartenant à des domaines connexes à l'application en cours de construction. Enfin, en haut de la hiérarchie, les évolutions de "mon domaine" sont spécifiques au domaine d'application choisi. In fine, le domaine caractérise la portée des évolutions, *i.e.* leur de degré de réutilisabilité. Nous pouvons recenser :

- Les évolutions généralistes ou indépendantes d'un domaine : ce sont des évolutions générales qui ne dépendent pas d'un domaine particulier. Ce type d'évolution est engendré par des réutilisations sur des architectures quelconques.
- Les évolutions dédiées ou dépendantes du domaine : ce sont des évolutions réutilisables à travers les applications d'un même domaine. Ce type d'évolution est engendré par des réutilisations au sein d'une famille d'architectures.
- Les évolutions orientées application : ce sont des évolutions spécifiques à une application donnée. Elles sont peu réutilisables. Ce type d'évolution est généralement engendré par des réutilisations ad-hoc non planifiées sur une architecture.

2.2.3.4 Niveau de modélisation

Comme nous l'avons vu au chapitre précédent, la description d'une architecture logicielle peut se faire au travers de trois niveaux de modélisation : le niveau méta-architecture (A2), le niveau

architecture (A1) et le niveau application (A0). Le niveau architecture est le niveau de description de famille d'applications en instanciant un ou plusieurs concepts du niveau méta. Le niveau application décrit le système en instanciant un ou plusieurs éléments du niveau architecture. Comme cela est mentionné fort justement par Medvidovic *et al.* [MR97], il existe des besoins d'évolution au moment de la spécification et au moment de l'exécution de l'architecture logicielle. Dans le cadre de la méta-modélisation, ces besoins correspondent respectivement à de l'évolution au niveau architecture et au niveau application. Nous pouvons ainsi identifier :

- L'évolution au niveau A1 : en considérant que les types de composants et les types de connecteurs sont instanciés à chaque fois qu'ils sont utilisés dans une architecture, leur évolution peut être vue simplement en terme de sous-typage. Par exemple, il peut être utile de faire évoluer un même composant de plusieurs façons, à travers différents mécanismes de sous-typage. L'évolution à ce niveau peut aussi prendre la forme de "refactorings" ou encore viser à introduire des motifs architecturaux pour améliorer la conception de l'architecture.
- L'évolution au niveau A0 : la modélisation explicite des architectures est destinée à supporter le développement et l'évolution de grands systèmes, ayant potentiellement une longue durée d'exécution. Être capable de faire évoluer de tels systèmes durant leur exécution peut être souhaitable et, dans certains cas, vital. L'évolution à ce niveau vise le dynamisme des architectures en permettant la réplication, l'insertion, le retrait, et la reconnexion d'éléments architecturaux ou de sous-architectures à l'exécution.

2.2.3.5 Mécanique opératoire

La mécanique opératoire concerne les dispositifs offerts pour spécifier et gérer l'évolution. On peut citer par exemple la possibilité de spécialiser, combiner ou encore instancier des évolutions. Ces mécanismes, quand ils existent, sont associés à un mode de raisonnement pour les faire fonctionner. Classiquement, on distingue le raisonnement déductif, inductif, abductif, classificatoire, analogique, etc. Relié à ce dernier point, le cœur de la mécanique opératoire peut être centralisé ou au contraire être distribué. Nous distinguons deux types de mécanique opératoire :

- Une mécanique basique : les évolutions proposées sont livrées telles quelles avec l'ADL. Toute la mécanique est "cablée" par avance.
- Une mécanique avancée : les évolutions doivent être écrites selon les besoins. Une mécanique est fournie dans ce but.

2.2.4 Qualité d'un modèle d'évolution

Le troisième et dernier niveau de la pyramide des besoins traite des qualités et des caractéristiques qui doivent être raisonnablement satisfaites par un modèle d'évolution, soit :

1. La réutilisabilité (R) : représente le degré de réutilisation offert par un modèle d'évolution.
2. Le support (S) : exprime la facilité de supporter l'évolution d'une application selon un modèle d'évolution donné.
3. L'adaptabilité (A) : exprime l'aptitude à contrôler et à permettre à un modèle d'évolution de s'adapter dynamiquement.
4. La performance (P) : exprime la vitesse d'exécution d'une évolution implémentée suivant un modèle d'évolution donné.

Supposons que $R(p)$, $S(p)$, $A(p)$, $P(p)$ représentent respectivement la réutilisabilité, le support, l'adaptabilité et la performance en fonction du degré de paramétrage p et α, β, χ, δ représentent les poids associés respectivement à $R(p)$, $S(p)$, $A(p)$, $P(p)$ pour favoriser un critère sur un autre. On peut alors noter que la qualité d'un modèle d'évolution donné ($\mathbf{Q}_{ME}$) peut être exprimée par :

$$\boxed{\mathbf{Q}_{ME} = \alpha R(p) + \beta S(p) + \chi A(p) + \delta P(p)}$$

Par ailleurs, pour accroître la réutilisabilité $R(p)$ d'un modèle d'évolution, certains critères sont préconisés comme l'extensibilité, l'évolutivité, la compositionalité et la remplaçabilité.

1. L'extensibilité : traduit le fait, par exemple, que l'ajout d'une évolution s'appliquant à un élément évolutif n'interfère pas avec les autres évolutions de cet élément.
2. L'évolutivité : signifie, par exemple, qu'une évolution est réutilisable et son fonctionnement peut être mise à jour sans impact sur les évolutions l'utilisant.
3. La compositionalité : est obtenue, par exemple, si une évolution inclut d'autres évolutions et délègue sa fonctionnalité à ses évolutions internes.
4. La remplaçabilité : est la possibilité de remplacer une évolution par une autre évolution qui offre la même fonctionnalité.

2.3 Étude des modèles d'évolution existants

Pour notre étude, nous retenons un certain nombre d'approches et leurs ADLs issus des communautés académiques américaine, européenne et française. A travers la manière de spécifier et de gérer l'évolution, chaque approche incarne un modèle d'évolution particulier. La sélection de ces approches a été basée sur leur notoriété (*i.e.*, les plus référencés dans la littérature) et/ou sur leur volonté affichée à traiter la problématique de l'évolution dans les architectures. En outre, la répartition des approches par école académique offre une classification transversale des travaux sur l'évolution architecturale. Dans les sections suivantes, pour chaque approche nous apportons :

- Une présentation de l'ADL auquel elle est associée, au travers notamment des principaux concepts structuraux de cet ADL,
- Une analyse des possibilités d'évolution offertes,
- Un positionnement vis-à-vis des trois niveaux de la pyramide des besoins.

2.3.1 Approches de l'école américaine

Nous présentons dans cette partie les approches d'évolution proposées par les ADLs C2, Dynamic Wright et Dynamic ACME.

2.3.1.1 C2

Strictement parlant, C2 est le paradigme de conception architecturale, alors que le nom de l'ADL associé est C2-SADL (Software Architecture Description Evolution Language) [MRT99]. Par souci de brièveté, nous ferons référence à ce langage simplement comme C2. Dans C2, les composants et les connecteurs sont tous deux des citoyens de première classe, et le passage de message est utilisé pour relier les composants entre eux à travers des connecteurs, sous réserve

de certaines restrictions. C2 supporte un mécanisme de composition hiérarchique pour créer des composants composites. En outre, les connecteurs peuvent effectuer le filtrage des messages, de manière facultative.

L'évolution dans C2 C2 permet de lier ("souder" dans le jargon C2) dynamiquement des composants avec les connecteurs, ainsi que de les délier. Toutes les possibilités de reconfiguration ou de recâblage d'une architecture sont autorisées : il est possible de délier un composant d'un autre, de le relier à un autre, de le laisser délié mais persistant dans le système, ou de le supprimer définitivement du système. Il est à noter que la caractéristique intéressante de C2 est que la reconfiguration dynamique est accomplie en utilisant le passage de message entre composants, au même titre que la communication ordinaire entre eux. D'autre part, un connecteur du style C2 est intrinsèquement évolutif du fait qu'il n'impose pas une interface spécifique, cette dernière s'adaptant en fonction des composants qui s'attachent et se détachent d'elle. Les reconfigurations sont spécifiées dans un (sous)langage de modification d'architecture (*Architecture Modification Language* – AML). Il s'agit d'un langage de script pouvant contenir les opérations suivantes : `addComponent`, `removeComponent`, `weld`, et `unweld`. On remarque que C2 ne supporte pas explicitement l'ajout et le retrait de connecteurs. A la place, ces opérations sont réalisées implicitement en fonction des liaisons et déliaisons. Par exemple, on considère la suppression tacite d'un connecteur lorsque celui-ci ne possède plus aucune liaison avec des composants. En outre, C2 propose aussi un mécanisme de sous-typage pour l'évolution des types de composants. Les auteurs dénombrent plusieurs déclinaisons de ce mécanisme, pouvant porter sur le nom, l'interface, le comportement ou l'implémentation. Le sous-typage hétérogène est alors la combinaison de ces sous-typages par les clauses *and*, *or* et *not*. Enfin, il est important de mentionner que C2 possède un opérateur particulier, `upgrade`, visant à améliorer un composant. Cette opération consiste à sous-typer le composant, modifier le sous-type, et remplacer les anciennes instances par les instances du nouveau type.

Positionnement de C2

Niveau 1 C2-SADL s'occupe de formuler les évolutions dans une architecture. La conservation de la trace des évolutions n'est pas gérée dans C2. On peut considérer que C2 possède une gestion d'impacts dans le cadre de son opérateur d'amélioration (`upgrade`), qui précise l'impact de l'évolution d'un type sur ses instances.

Niveau 2 C2-SADL adresse le niveau spécification des architectures qu'il décrit, mais propose à travers l'environnement DRADEL un raffinement vers du code Java. A l'aide d'AML, l'architecte exploite au sein d'un script des primitives dont il ne connaît pas le fonctionnement interne. De plus, le script obtenu ne constitue pas une nouvelle évolution réutilisable. C2-SADL est entièrement dédié aux systèmes logiciels qui emploient le style architectural C2. La plupart des GUI (*Graphic User Interface*) et les applications distribuées sont structurés autour du style C2 [TMA*95]. Cette dépendance au style C2 s'exprime par exemple à travers les primitives `weld/unweld` qui fonctionnent implicitement sur des ports et rôles de type *bottom* et *top* définis par le style C2. Nous l'avons vu, C2 supporte l'évolution au niveau des types et au niveau des instances, voire les deux simultanément via l'opérateur d'amélioration.

Niveau 3 Le modèle d'évolution choisi par C2 ne permet pas la réutilisatibilité ni l'adaptabilité, mais la mise en œuvre dans ce langage est plutôt aisée.

2.3.1.2 Dynamic Wright

Wright [All97] permet d'analyser à la fois l'architecture de systèmes logiciels individuels et de familles de systèmes. Wright a aussi exploré la nature des abstractions architecturales en elles-même. En particulier, le travail sur Wright s'est concentré sur le concept de type de connecteur explicite, sur la validation automatique de propriétés architecturales, et sur la formalisation des styles architecturaux. L'ADL Wright est construit autour de trois principaux concepts architecturaux : les composants, les connecteurs et les configurations.

L'évolution dans Dynamic Wright Wright n'a pas été construit au départ pour spécifier et analyser la dynamique d'une architecture logicielle. Cependant, une approche proposée dans [ADG98] le permet. Le résultat, dénommé Dynamic Wright, inclut certaines fonctionnalités pour aider à décrire le dynamisme. Dynamic Wright représente la structure d'une architecture comme un graphe dans lequel les composants et les connecteurs sont des noeuds, et spécifie le comportement et la reconfiguration d'un système dans une variante de l'algèbre de processus. En fait, Wright a adopté une approche similaire à la reconfiguration conditionnelle proposée par l'ADL Rapide [LV95] : il fait la distinction entre la communication et les événements de contrôle, où ces derniers sont utilisés pour spécifier des conditions sous lesquelles les changements dynamiques sont autorisés [ADG97]. Le "Configurateur" est responsable de l'application des changements à la configuration architecturale par le biais des actions de création ou de destruction de composants et de connecteurs (`new`, `del`) et de création ou de destruction de liaisons (`attach`, `detach`).

Positionnement de Dynamic Wright

Niveau 1 Dynamic Wright adresse la formulation des évolutions, mais ne traite pas la gestion des impacts ni la trace de l'évolution.

Niveau 2 Wright peut être utilisé pour fournir une signification précise et abstraite à une spécification architecturale. Un travail important a été mené afin d'analyser un assemblage de composants, mais les problématiques liées à la génération de code ont été ignorées. Dynamic Wright met l'accent sur la spécification du comportement dynamique : création, suppression, interconnexions qui se modifient dynamiquement. Des éléments de reconfiguration, constituant un vocabulaire de contrôle, ont pour cela été ajoutés au vocabulaire de base. Les règles de reconfiguration définies par l'architecte ne peuvent pas être réutilisées, et ce dernier ne peut pas connaître ni modifier le fonctionnement prédéfini des actions de création ou de destruction. Dans Dynamic Wright, le configurateur centralise le raisonnement en sélectionnant les règles à appliquer en fonction de propriétés, etc. Par ailleurs, Wright s'est intéressé à la notion de style architectural en distinguant les types des composants et des connecteurs de leurs instances. Ces types peuvent évoluer par un mécanisme de composition hiérarchique.

Niveau 3 Le modèle d'évolution proposé par Dynamic Wright n'est ni réutilisable, ni adaptable et repose sur des notations difficiles d'accès pour un architecte novice.

2.3.1.3 Dynamic Acme

ACME est considéré comme un ADL pivot (ou ADL d'échange [GMW97]) capable d'incorporer les fonctionnalités des autres ADLs, permettant ainsi aux utilisateurs d'exprimer des choses qu'ils auraient pu exprimer dans n'importe lequel des autres langages. ACME représente un plus petit dénominateur commun qui inclut les aspects des descriptions architecturales partagés par tous les ADLs. ACME offre des constructions pour décrire des architectures (appelées systèmes) comme des graphes de composants et de connecteurs, un mécanisme de représentation pour la décomposition hiérarchique des composants et des connecteurs en sous-systèmes, ainsi qu'un moyen de décrire des familles et des types d'éléments sur la base de ces constructions. Pour assurer son statut de langage d'échange, les éléments peuvent être annotés par des propriétés qui représentent des informations descriptives (*e.g.*, comportement, code source) auxquelles les outils ou personnes devront donner une interprétation adéquate.

L'évolution dans Dynamic Acme ACME ne fournit aucun moyen de spécifier de manière simple et claire la dynamique d'un système. En effet, le système est décrit comme un ensemble d'instances de composants reliées entre elles par des connecteurs. Cependant, cette description de l'assemblage reste figée. Aucun moyen n'est fourni pour spécifier les changements de l'application au cours de son éxécution. Dynamic ACME [Wil01a] est une extension à ACME pour modéliser les architectures dynamiques. D'une part, Dynamic ACME distingue deux catégories d'éléments architecturaux, ceux ouverts aux modifications (mot clé `open`), et ceux qui ne le sont pas. Les éléments ouverts peuvent être sous-typés par exemple. D'autre part, Dynamic ACME introduit les notions d'éléments optionnels et d'éléments multiples, qui jouent le rôle de quantificateurs sur le nombre d'instances possibles. Un outil a été proposé pour assister l'architecte, nommé AcmeStudio. Malheureusement, l'outil n'a pas suivi la progression du langage et les nouveautés de Dynamic ACME n'ont pas été intégrées.

Positionnement de Dynamic Acme

Niveau 1 Dynamic Acme s'intéresse à la formulation de l'évolution uniquement.

Niveau 2 En tant qu'ADL pivot, ACME ne s'intéresse pas aux problématiques de niveau code et du raffinement d'une architecture décrite en ACME. Dynamic ACME est d'avantage conçu pour maîtriser l'évolution que pour la spécifier, et se focalise sur le niveau des types de composants et de connecteurs. D'ailleurs, les insuffisances d'ACME en terme de reconfiguration dynamique ont donné naissance à des extensions comme dans [CGS*02] ou [BJC05]. Les quelques mécanismes d'évolutions proposés par ACME/Dynamic ACME sont "cablés" dans l'ADL et ne permettent que des évolutions de fine granularité.

Niveau 3 Le modèle d'évolution de Dynamic Acme n'est ni réutilisable, ni adaptable mais les constructions d'évolution du langage sont simples à appréhender.

2.3.2 Approches de l'école européenne

Nous présentons dans cette partie les approches d'évolution proposées par les ADLs Darwin, xADL 2.0 et SOFA.

2.3.2.1 Darwin

Darwin [MK96] supporte les composants mais pas les connecteurs en tant que citoyens de première classe (c'est-à-dire des connecteurs implicites). Les composants sont instanciés à partir de leurs types, et le sous-typage peut être utilisé pour construire des composants plus spécifiques à partir de composants génériques. Chaque type de composant est décrit en termes de services qu'il fournit et de services dont il a besoin. Darwin permet aussi de définir des composants composites obtenus à partir de composants simples. Les services sont spécifiés en utilisant le π-calcul, tandis que la configuration est spécifiée en liant les services requis par un composant aux services fournis par un autre composant, en utilisant l'opérateur `bind`. Cette liaison ne peut avoir lieu que si les types des services mis en jeu sont compatibles.

L'évolution dans Darwin Darwin permet aux composants d'être instanciés, et leurs services liés, soit statiquement soit à l'exécution. Dans ce dernier cas, deux options sont disponibles. Avec l'option d'instanciation paresseuse, un élément n'est pas instancié jusqu'à ce qu'un autre composant demande à utiliser ses services. Toutefois, les types des composants participants et les liaisons entre eux doivent être spécifiés au préalable. De cette manière, le système peut évoluer au moment de l'exécution, mais la manière dont celui-ci évolue est fixé lors de la conception. L'autre option, connue sous le nom d'instanciation dynamique directe, permet à des structures d'évoluer de façon arbitraire au moment de l'exécution. Toutefois, si les composants instanciés dynamiquement désirent interagir, les options disponibles sont limitées. Notamment, si ces éléments souhaitent interagir directement, ils doivent échanger les informations pertinentes (c'est-à-dire, les références aux services) à travers une tierce partie, qui n'appartient pas au langage en soi mais à sa plateforme d'exécution nommée Regis. Dans les deux cas, les liaisons sont permanentes et ne peuvent être défaites. En d'autres termes, les architectures spécifiées dans Darwin peuvent grossir, mais elles ne peuvent pas diminuer [3].

Positionnement de Darwin

Niveau 1 La formulation de l'évolution fait partie des préoccupations de Darwin, contrairement à la conservation de la trace de l'évolution. La dynamique dans Darwin repose sur les schémas d'instanciation décris au préalable. On peut voir dans cette approche une forme de gestion des impacts puisque il n'est pas nécessaire de spécifier explicitement les connexions entre des composants nouvellement créés par exemple. A chaque instanciation d'un composant, et conformément au schéma d'instanciation donné, Darwin créé automatiquement les connexions nécessaires.

Niveau 2 Darwin accepte un raffinement des spécifications vers la plateforme Régis qui fournit un cadre écrit en C++. L'évolution dans Darwin est supportée au niveau des types ou des instances. Dans les deux cas, l'architecte passe par des constructions non modifiables et de fine granularité.

Niveau 3 Le modèle d'évolution incarné par Darwin ne favorise pas la réutilisabilité, pas plus que son adaptabilité. D'autre part, l'approche par schéma d'instanciation rend peu naturel la spécification et la gestion de l'évolution dans l'architecture.

3. Peut tout de même se faire au niveau langage de programmation

2.3.2.2 xADL 2.0

xArch est une représentation commune basée sur le format XML pour la description d'architecture qui peut être étendue à d'autres ADLs existants (voir les extensions à ACME, C2 et Darwin proposées dans [DRM00]). Il contient les éléments primitifs qui composent une description d'architecture d'un point de vue structurel. xADL 2.0 est un ADL basé sur xArch qui est défini comme un ensemble de schémas XML [DHT01].

L'évolution dans xADL 2.0 Parmi les schémas proposés par xADL, le schéma de conception architectural (*Structure and Types Extension*) sépare la description architecturale en deux blocs : les types et la structure. Le premier permet de décrire les types (possiblement composites) de composants, de connecteurs et d'interfaces tandis que le second permet de prescrire les instances de ces types dans une architecture. L'évolution est disponible par l'intermédiaire de l'outil ArchStudio 4, où il est possible d'ajouter ou de retirer des types et des instances d'éléments architecturaux. Ceci se fait de manière transparente grâce à une API pour la manipulation de documents XML. Grâce à des schémas spécifiques, l'évolution dans xADL prend également la forme d'une gestion de configuration traditionnelle, comme la spécification des options, des versions et des variantes, appliquée aux éléments architecturaux et à l'architecture elle même.

Positionnement de xADL 2.0

Niveau 1 xADL 2.0 traite de la formulation des évolutions d'une part, et de la conservation de la trace de l'évolution d'autre part. Ce dernier point repose sur les notions de versions, de variantes et d'options sur les types d'éléments architecturaux. Il existe par ailleurs MAE [RHMRM04], un environnement construit autour de xADL 2.0 et offrant un système complet de gestion de versions.

Niveau 2 ArchStudio 4, l'environnement de développement associé à xADL2.0 s'appuie sur une technique de Data Binding [4] pour gérer le raffinement. Le Data Binding consiste à produire du code orienté objet (en l'occurrence du code Java) à partir des items XML ; ceci en dérivant de façon systématique des classes et des interfaces en correspondance avec le contenu des schémas. En soi, xADL 2.0 est un ADL généraliste. Cependant, il est conçu dès le départ pour être spécialisé par les utilisateurs afin d'optimiser le langage pour des domaines particuliers (*e.g.*, une extension a été créée par le Jet Propulsion Laboratory de la NASA). Néanmoins, les évolutions pré-définies par ArchStudio elles, restent généralistes. Le fonctionnement de ArchStudio – et à travers lui des opérations d'évolution – est opaque et les scénarios d'évolution construits de la sorte ne sont pas capitalisables pour une réutilisation ultérieure.

Niveau 3 Le modèle d'évolution proposé par xADL 2.0 ne tient pas compte de la réutilisabilité et de l'adaptabilité. Toutefois, l'utilisation du formalisme XML et d'un outil graphique offre un bon support pour l'évolution.

4. xArch/xADL 2.0 Data Binding Library

2.3.2.3 SOFA

SOFA [PBJ98] est un projet visant à offrir une plateforme pour les composants logiciels. Dans SOFA, les applications sont créées en assemblant des composants et des connecteurs. SOFA est un modèle hiérarchique, c'est-à-dire que les composants peuvent eux-même être composés d'autres composants ou bien ils peuvent être des briques de base contenant directement une implémentation. Le composant ou le connecteur est décrit par sa "frame" (vue boite noire) et son "architecture" (vue boite grise). Ces concepts séparent respectivement la notion de type de la notion d'instance. Notons au passage que le nom exact de l'ADL dans SOFA est CDL (*Component Definition Language*), basé sur IDL (*Interface Definition Language*) défini par l'OMG.

L'évolution dans SOFA Une application au sens SOFA contient une liste des instances des composants et des connecteurs ainsi que leurs liaisons. Même si il est possible de remplacer un composant à chaud par le bais de la plateforme d'exécution DCUP, la description de l'assemblage ne peut pas être modifiée. Cependant, SOFA a mis l'accent sur la gestion de versions. Un entrepôt (le *template repository*) contient les implémentations des composants (binaires) et leur description abstraite (CDL). L'entrepôt dispose d'un mécanisme de gestion de version permettant d'y ranger différentes versions des composants. Ce système est très intéressant au niveau du suivi de l'évolution d'une architecture logicielle. Une extension de cet entrepôt et de ses mécanismes est par ailleurs proposée dans [MH01].

Positionnement de SOFA

Niveau 1 SOFA traite de la formulation de l'évolution et se focalise sur la gestion de versions pour permettre de garder une trace d'une évolution liée à l'action de l'architecte. Les impacts ne sont pas traités dans SOFA.

Niveau 2 SOFA propose la spécification d'une architecture adaptée à sa mise en oeuvre de référence : DCUP (uniquement en Java pour le moment). En effet, SOFA est issu d'une équipe qui a beaucoup travaillé dans le domaine des objets répartis et qui s'est logiquement attachée à offrir une structure d'accueil correspondant au modèle de composant de SOFA. SOFA propose une approche statique de la description d'une architecture logicielle comme un assemblage de composants à un instant *t*. Ainsi, SOFA gère l'évolution des types de composants et de connecteurs mais pas celui des instances. Pour faire évoluer ses types, l'architecte se repose sur des évolutions de type boites noires, sans possibilités de réaliser des assemblages d'évolutions. Dans ce cadre, la mécanique opératoire est basique.

Niveau 3 Le modèle d'évolution promu par SOFA n'est pas réutilisable ni adaptable mais l'utilisation du mécanisme de gestion de version est assez simple et bien outillé.

2.3.3 Approches de l'école française

Nous présentons dans cette partie les approches d'évolution proposées par Archware, SAEV et TranSAT.

2.3.3.1 ArchWare

Le but du projet ArchWare est de fournir un environnement de développement centré architecture pour la construction de systèmes évolutifs. La famille de langages ArchWare ADL propose, entre autres, un langage composant-connecteur appelé ArchWare C&C-ADL. Le modèle de composant sous-jacent est hiérarchique (*i.e.*, distinction entre composant atomique/composite) et traite les composants et les connecteurs comme des éléments de première classe.

L'évolution dans ArchWare Les architectes peuvent utiliser les capacités de ArchWare C&C-ADL pour représenter des architectures dynamiques [MKB*04]. L'évolution d'une architecture ou sous-architecture est prise en compte par un élément dédié, le "chorégraphe". Ce dernier est en charge de changer la topologie en cas de besoin : changer les attachements entre éléments architecturaux, créer dynamiquement de nouvelles instances, exclure des éléments de l'architecture, en inclure d'autres, etc. La spécification de la partie chorégraphique s'ajoute à la spécification de la configuration initiale de l'architecture. On peut utiliser dans cette spécification les opérations `new` (composant et connecteur), `attach` et `detach`. En outre, ArchWare peut aussi traiter l'évolution non prévue dans une architecture, en intégrant un composant spécial dédié à l'évolution et en s'appuyant sur une machine virtuelle. Ce composant particulier nommé "evolver" peut être sollicité par une architecture lorsqu'une évolution doit avoir lieu, et répond en faisant transiter des éléments architecturaux comme n'importe quelle donnée. Un pré-requis est que ces éléments aient été décrits dans un fichier qui est alors chargé dynamiquement par la machine virtuelle.

ArchWare inclut un langage nommé ARL (*Architecture Refinement Language*) dédié au raffinement d'architectures. Parmi les deux modes de raffinement supportés, le raffinement horizontal est utilisé pour inclure ou exclure des éléments d'une spécification d'origine. Ainsi, ARL fournit un ensemble d'opérations appelées actions de raffinement pour l'évolution d'architecture. ArchWare inclue également ASL (*Architecture Style Language*), un langage dédié à la définition de styles architecturaux en fournissant des types de composants et de connecteurs, ainsi que leurs contraintes. ASL met en place deux mécanismes d'évolution sur les styles : l'héritage et l'agrégation.

Positionnement de ArchWare

Niveau 1 ArchWare s'occupe de la formulation de l'évolution uniquement.

Niveau 2 Certes ARL supporte un raffinement vertical, mais celui-ci est cantonné au niveau de la spécification de l'architecture. L'évolution des types, comme l'évolution des instances est supportée dans C&C ADL. Dans ce dernier cas, l'utilisation du chorégraphe repose sur une gestion centralisée de l'évolution. Malgré un langage consacré à la définition de styles architecturaux et donc de familles d'architectures, les évolutions proposées sont de natures généralistes. En outre, les évolutions décrites ne sont pas réutilisables dans des stratégies plus complexes, et les primitives proposées sont de type boites noires.

Niveau 3 Le modèle d'évolution de ArchWare n'offre pas la réutilisabilité ni l'adaptabilité. Les notations sur lesquelles repose le modèle d'évolution de ArchWare sont assez fastidieuses.

2.3.3.2 SAEV

SAEV est un modèle d'évolution générique, uniforme et indépendant de tout ADL, permettant la spécification et la gestion de l'évolution des architectures logicielles [SH07]. L'idée est d'associer des stratégies d'évolution à chaque élément d'une architecture. Une stratégie d'évolution sur un élément regroupe des règles d'évolution le concernant, c'est-à-dire les opérations applicables à ce dernier. Une règle d'évolution est déclenchée lorsqu'un événement précis survient et selon certaines conditions (on parle de règles actives).

L'évolution dans SAEV L'évolution dynamique dans SAEV repose sur un ensemble de règles définies par l'architecte, et associées aux différents éléments architecturaux. Puisque que la démarche de méta-modélisation est présente dans SAEV, l'évolution statique est prise en compte en ré-appliquant les mêmes mécanismes au niveau de modélisation supérieur. La description des règles dans SAEV repose sur le formalisme ECA (Événement-Condition-Action). Ainsi, chaque règle est composée d'un évènement, d'une condition et d'une action. La partie évènement représente le message d'invocation pouvant être émis par l'architecte ou par une autre règle. La partie condition exprime ce qui doit être nécessairement vérifié pour pouvoir exécuter la règle. Enfin, la partie action, décris une opération d'évolution simple (ajout, modification, suppression) ou des événements afin de déclencher d'autres règles. Ainsi, le chaînage avant des règles permet de déterminer les impacts à propager.

Positionnement de SAEV

Niveau 1 SAEV a pour objectif de formuler les évolutions et de gérer leurs impacts. SAEV s'est concentré sur la gestion des impacts en supportant l'enchaînement des règles par l'intermédiaire d'émissions et de réceptions d'évènements de même nature.

Niveau 2 SAEV se veut indépendant d'un ADL particulier et l'aspect implémentation a logiquement été reléguée au second plan. Le fonctionnement de SAEV nécessite un moteur de règles, appelé "gestionnaire d'évolution", qui centralise l'ensemble du raisonnement. Chaque règle peut être modifiée selon les besoins, soit dans sa partie évènement et condition, soit dans sa partie action. Les règles peuvent concerner aussi bien des types d'éléments architecturaux que leurs instances. Il est à noter qu'en plus du formalisme ECA, les auteurs ont proposé une notation graphique des règles dans l'environnement AGG [Tae00]. Dans cet environnement, les architectures sont représentées par des graphes et chaque règle est représentée implicitement en définissant deux morceaux de graphe : une partie gauche (LHS - *Left Hand Side*) décrivant la situation avant évolution et une partie droite (RHS - *Right Hand Side*) décrivant la situation après évolution.

Niveau 3 Le modèle d'évolution de SAEV permet la réutilisabilité mais pas l'adaptabilité. La simplicité reconnue du formalisme ECA offre un bon support pour l'évolution.

2.3.3.3 TranSAT

TranSAT [BD04] est considéré comme un canevas de conception d'architecture qui permet l'intégration de nouvelles préoccupations dans une architecture par transformation de cette der-

nière. L'architecture de base est spécifiée à l'aide de SafArchie ADL, reposant sur cinq concepts architecturaux (composant composite, composant primitif, liaison, port, opération) ainsi que sur la notion de contrat. TranSAT propose un langage dédié pour spécifier les modifications à apporter sur l'architecture de base afin d'intégrer la nouvelle préoccupation.

L'évolution dans TranSAT Inspiré par les technologies des aspects, TranSAT introduit la notion de patron d'architecture pour structurer les différentes préoccupations transverses d'une architecture. Ce patron comprend les éléments à intégrer, les transformations à apporter à l'architecture de base, mais aussi un ensemble de contraintes génériques sur les éléments d'une architecture cible sur laquelle le patron peut être intégré. En particulier, le langage offert pour exprimer les règles de transformation adresse l'évolution structurelle à travers un ensemble de douze primitives. Il s'agit d'opérateurs d'ajout, de suppression et de déplacement appliqués aux composants (primitifs ou composites) et aux liaisons, ainsi qu'aux ports et à leurs opérations. Volontairement, aucune structure de contrôle n'est fournie pour combiner les règles afin de permettre une analyse statique de l'impact de l'intégration de la préoccupation. Ces règles sont regroupées dans un script qui sera utilisé – entre autres choses – par un moteur de transformation nommé "tisseur" pour produire une nouvelle description d'architecture logicielle.

Positionnement de TranSAT

Niveau 1 TranSAT permet de formuler l'évolution, mais ne gère pas les impacts, pas plus qu'il ne garde la trace de l'évolution.

Niveau 2 Clairement, TranSAT travaille au niveau de la spécification d'une architecture. Les primitives à la disposition de l'architecte sont fournies avec le tisseur sans qu'il soit possible d'en modifier ni de comprendre le fonctionnement. Par ailleurs, les transformations ne sont pas réutilisables, c'est-à-dire que des transformations ne peuvent pas faire appel à d'autres transformations. Il est important de noter que les règles de transformation ne s'occupent pas de la hiérarchie des composants dans l'architecture de base, laissant au tisseur la charge de créer les bons éléments aux bons endroits. En somme, le moteur de transformation centralise le raisonnement pour déterminer où appliquer les règles, et pour les exécuter.

Niveau 3 Le modèle d'évolution proposé par TranSAT ne cible pas la réutilisabilité ni l'adaptabilité et son approche à base d'aspect n'est pas des plus évidentes malgrè de bons résultats.

2.3.4 Bilan de l'étude

Nous avons présenté dans les sections précédentes les différents modèles d'évolution proposés par la littérature pour tenter de faire face à la problématique de l'évolution dans les architectures logicielles. Dans cette section, nous proposons de tirer le bilan des modèles étudiés en s'appuyant sur la pyramide des besoins à trois niveaux. A la lumière de ces éléments, nous analyserons l'existant afin d'en extraire les points forts et les points faibles.

2.3.4.1 Récapitulatif

Les tableaux 2.2, 2.3 et 2.4 récapitulent respectivement les approches étudiées selon les niveaux croissants de la pyramide des besoins. Comme la performance dépend non seulement de la structure d'un modèle, mais également d'autres facteurs tels que l'application cible ou le système sous-jacent, nous ne l'avons pas inclus dans la comparaison du niveau qualité. Une analyse de ce récapitulatif et une discussion est proposée dans la section suivante.

	NIVEAU 1 : INTENTION DU MODELE D'EVOLUTION		
	Formuler l'évolution	***Gérer les impacts de l'évolution***	***Garder la trace de l'évolution***
C2	√	√ (upgrade)	
Dyn. Wright	√		
Dyn. ACME	√		
Darwin	√	√ (Politiques d'instanciation)	
xADL 2.0	√		√ (Versions, variantes et options)
SOFA	√		√ (Template Repository)
ArchWare	√		
SAEV	√	√ (chainage de règles)	
TranSAT	√		

Table 2.2 – Positionnement des modèles d'évolution selon la base de la pyramide des besoins.

2.3.4.2 Analyse et discussions

Niveau 1 Tous les modèles d'évolution étudiés s'occupent de la formulation de l'évolution. En revanche, peu d'entre eux ont proposé une gestion des impacts, ou alors celle-ci était limitée à quelques cas bien définis. Dans l'ensemble, les modèles étudiés n'ont pas cherché à garder la trace de l'évolution.

Niveau 2 Par nature, les ADLs académiques n'ont pas vocation à assurer le raffinement des spécifications vers le code source. Par conséquent, peu d'entre eux cherchent à refléter les évolutions du niveau architectural sur le niveau code. Directement relié à ce dernier point, les modes d'expression prennent souvent la forme de formalismes de haut-niveau.

Toutes les approches étudiées fournissent un moyen de spécifier un assemblage de primitives d'évolution. Ces spécifications peuvent être contenues à l'intérieur de l'ADL (partie orchestrateur, partie configurateur, etc.) ou à l'extérieur de l'ADL (scripts, transformations, etc.). Très peu d'ADL permettent de considérer de telles spécifications comme de nouvelles opérations d'évolution. En d'autres termes, les seules évolutions qu'il est possible de réutiliser sont de fine granularité.

A quelques exceptions près, les évolution proposées appartiennent à la couche tout domaine, c'est-à-dire utilisables sur une architecture quelconque. A première vue, cela peut paraître satisfaisant. Pourtant, certaines familles d'applications nécessitent de prendre en compte leurs spécificités dans leurs évolutions. Ces dernières nécessitent des opérations décrites dans les termes du vocabulaire d'un style architectural et qui devraient aboutir à des architectures qui satisfont ce style. Par exemple, une opération pour ajouter un client dans le style client/serveur doit aussi

	NIVEAU 2 : EXPRESSIVITE DU MODELE D'EVOLUTION				
	Niveau d'abstraction	***Mode d'expression***	***Domaine***	***Niveau de modélisation***	***Mécanique opératoire***
C2	- Spécification / Boite noire - Code Java (DRADEL) / Boite noire	- AML / Grain fin	- Dédié au style C2	- A1 : Sous-typage hétérogène et composition hiérarchique de types de composants. - A0 : ajout/suppression de composants - A1/A0 : Amélioration de composant	Basique
Dyn. Wright	- Spécification / Boite noire	- Règles+CSP / Grain fin	- Tout domaine	- A1 : Composition hiérarchique de types de composants et de connecteurs - A0 : ajout/suppression de composants et de connecteurs	Basique (raisonnement déductif)
Dyn. ACME	- Spécification / Boite noire	- Sur-ensemble du langage ACME / Grain fin	- Tout domaine	- A1 : Sous-typage et composition hiérarchique de types de composants et de connecteurs. Mot clé Open. - A0: Multiplicité pour contrôler le nombre d'instances	Basique
Darwin	- Spécification / Boite noire - Code C++ (Regis)	- Schéma d'instanciation préalable / Grain fin	- Tout domaine	- A1 : Sous-typage et composition hiérarchique de composant - A0 : Instanciation dynamique et paresseuse	Basique
xADL 2.0	- Spécification / Boite noire - Code Java (DataBinding) / Boite noire	- Transformation arbre XML (ArchStudio 4) / Grain fin	- Tout domaine	- A1 : Versions, variantes, options, ajout, retrait de types de composants et de connecteurs. - A0 : ajout/retrait de composants et connecteur	Basique
SOFA	- Spécification / Boite noire - Code Java (DCUP)/ Boite noire	- Ad-hoc / Grain fin	- Tout domaine	- A1 : Composition hiérarchique de types de composants. Gestion de version.	Basique
ArchWare	- Spécification / Boite noire - Code (Raffinement vertical)	- Règle, ASL, ARL / Grain fin	- Tout domaine	- A1 : Héritage et agrégation de styles architecturaux - A0 : Raffinement horizontal. Chorégraphe. Composant « Evolver »	Basique (raisonnement déductif)
SAEV	- Spécification / Boite blanche	- Règle ECA / Grain fin-fort - Règle LHS/RHS (AGG) / Grain fin-fort	- Libre	- A1 : Libre - A0 : Libre	Avancée (raisonnement déductif)
TranSAT	- Spécification / Boite noire	- Langage de transformation dédié / Grain fin	- Tout domaine	- A1 : composition hiérarchique de types de composants. - A0 : ajout/suppression/déplacement de composants/ports/liaisons/opérations	Basique (raisonnement déductif)

Table 2.3 – Positionnement des modèles d'évolution selon le second niveau de la pyramide des besoins.

connecter ce dernier à un serveur. De même, le retrait d'un serveur peut transférer ou supprimer ses clients.

Les ADLs de première génération, notamment Wright et Darwin, ne se sont pas préoccupés de la problématique de l'évolution au moment de la spécification de l'architecture logicielle. Néanmoins, nous avons identifié certains mécanismes offerts par ces ADLs qui permettent l'évolution au niveau A1, comme le sous-typage ou la composition hiérarchique de types de composants et de connecteurs. On notera au passage que la notion de sous-typage adoptée par les ADLs est plus riche que celle proposée par les langages de programmation. La majorité des ADLs de seconde génération réifient et considèrent comme entités de première classe tous les concepts de base d'une description d'architecture logicielle : composant, connecteur, interface et configuration. Ces éléments sont autant d'éléments susceptiblent d'évoluer, aussi bien au niveau A1 que A0. Notons tout de même que la problématique globale de l'évolution au moment de l'exécution, qui inclut par exemple la question du transfert d'état des éléments architecturaux, n'est pas abordée

	NIVEAU 3 : QUALITE DU MODELE D'EVOLUTION					
	Réutilisabilité				*Support*	*Adaptabilité*
	Extensibilité	*Evolutivité*	*Compositionnalité*	*Remplaçabilité*		
C2					Bon	
Dyn. Wright					Faible	
Dyn. ACME					Moyen	
Darwin					Faible	
xADL 2.0					Bon	
SOFA					Moyen	
ArchWare					Moyen	
SAEV	√	√	√	√	Bon	
TranSAT					Moyen	

Table 2.4 – Positionnement des modèles d'évolution selon le dernier niveau de la pyramide des besoins.

directement dans les ADLs, mais plutôt au niveau de l'infrastructure d'accueil, quand elle existe (*e.g.*, DCUP pour SOFA).

Compte tenu des remarques précédentes, on peut considérer que la mécanique opératoire offerte par les modèles d'évolution étudiés est basique. On remarquera que plusieurs approches ont choisi une approche à base de règles pour permettre de gérer l'évolution d'une architecture. Historiquement, Rapide [LV95] a été un des premiers ADL à utiliser les règles pour spécifier le dynamisme. Ce type d'approche se retrouve récemment dans les travaux sur les architectures auto-adaptatives, basées sur des règles de type Observation–Réponse [GT04].

Niveau 3 Les approches étudiées sont avares en termes de mécanismes pour réutiliser les opérateurs d'évolution (*e.g.*, extension et composition). Spécifiquement, les propriétés d'évolutivité et de remplaçabilité ne sont pas favorisées. Par ailleurs, les ADLs formels reposent sur des théories mathématiques assurant l'obtention de spécifications non ambiguës. Ces formalismes peuvent paraître rebutants et ne sont pas aisément appréhendables par les non-initiés. En outre, les approches sont souvent ad-hoc. Si les besoins changent, il faut fournir de nouveaux outils. Aucun ADL ne prévoit de mécanisme de paramétrage (*template*) par exemple. Enfin, la performance dépendra fortement de la plateforme technologique cible et de la catégorie du langage utilisé : compilé ou interprété (moins performant). Pour les quelques ADLs qui ciblent l'implémentation, le langage choisi est un langage compilé.

2.4 Conclusion

Nous avons abordé dans ce chapitre les approches d'évolution au niveau architectural des systèmes logiciels au travers d'un ensemble d'ADLs et de propositions académiques. Dans un premier temps, nous avons donné un éclairage sur le bien-fondé de l'évolution basée architecture. Puis, nous avons proposé notre propre cadre de comparaison des modèles d'évolution organisé autour d'une pyramide des besoins à trois niveaux : (1) l'intention, (2) le pouvoir expressif, et

(3) la qualité du modèle. Ce cadre offre un référentiel [5] commun et indépendant d'une approche particulière.

Ainsi, en se basant sur ce référentiel nous avons étudié un ensemble de travaux issus des écoles américaine, européenne et française. Concrètement, nous avons décrit un certain nombre de modèles d'évolution suivant les trois niveaux de besoins que nous avons identifié. Ceci nous a permis de distinguer, pour chaque modèle, les aspects de l'évolution qui sont traités et ceux qui ne le sont pas.

Nous avons conclu cette étude en apportant notre propre évaluation des modèles d'évolution existants. En se basant sur cette dernière, nous avons pu dresser un bilan récapitulatif. Ce qui ressort essentiellement de ce bilan, c'est que les modèles d'évolution existants ne proposent pas de spécifier et de gérer l'évolution de manière explicite. Notamment, la notion de réutilisation de l'évolution est très peu abordée. Or, notre motivation pour faire face à la problématique de l'évolution architecturale est entièrement basée sur la notion clé de réutilisation, comme cela a été mis en évidence à la fin du Chapitre 1. Sur ce constat, notre objectif est de proposer un modèle d'évolution alternatif pour s'attaquer à cette problématique.

Dans le chapitre suivant, nous présentons un nouveau modèle d'évolution, il s'agit de SAEM.

5. Le lecteur intéressé par ces questions de comparaison trouvera dans l'Annexe A une étude de trois autres référentiels proposés dans la littérature.

Chapitre 3

Modèle d'évolution architecturale à base de style

Les deux principaux résultats du cycle du développement centré architecture sont les suivants : (i) l'architecture bien sûr mais également (ii) la connaissance de l'équipe d'architectes. Nous l'avons vu, un grand effort de recherche a été porté sur les langages de description d'architectures (ADL), pour rendre explicite de façon formalisée l'architecture des systèmes logiciels. En revanche, peu de recherche a été faite sur le savoir-faire des architectes eux-mêmes. Nous avons montré en section 1.1.2.2 qu'ils requièrent des aptitudes différentes de celles des concepteurs et des programmeurs, et il est clair qu'un "bon" architecte est quelqu'un de talentueux dont une grande partie des connaissances est tacite. Néanmoins, leurs bonnes pratiques dans le domaine de l'architecture peuvent être formalisées, et le travail à base de patrons ou de styles (*i.e.*, des représentations des façons standard de faire des choses) est très prometteur. Ceci est particulièrement efficace lorsque l'architecture n'est pas innovante, mais très similaire à des cas existants.

Par analogie, nous tendons à penser que l'évolution d'architectures, assurée par des architectes compétents, peut être formalisée. De la même façon, cette approche est particulièrement profitable lorsque les évolutions sont récurrentes ou similaires d'une architecture à une autre. C'est aussi un moyen de rentabiliser l'activité d'évolution, alors même que cette dernière constitue toujours un coût faramineux dans le budget des projets. Nous sommes convaincus que des bonnes pratiques d'évolution gagneraient à être capitalisées et partagées par la communauté des architectes. Idéalement, une solution à un problème d'évolution rencontré par un architecte sur un projet devrait être immédiatement mise à disposition des autres architectes concernés par des projets similaires, afin de leur éviter de réinventer la roue à chaque fois.

Dans ce chapitre nous introduisons notre proposition baptisée SAEM (Style-based Architectural Evolution Model), un nouveau modèle d'évolution dans les architectures logicielles à base de composants, reposant sur le concept clé de *style d'évolution*. En premier lieu, nous expliquons pourquoi et comment la notion de style peut être un instrument puissant de capitalisation de savoir et de savoir-faire pour l'évolution architecturale. Puis, nous présentons le méta-modèle de SAEM et nous en décrivons les différents concepts. Nous présentons également un formalisme de représentation des styles d'évolution baptisé MY. Nous terminons ce chapitre en procédant à l'évaluation de notre propre modèle d'évolution dans la pyramide des besoins présentée au chapitre précédent.

3.1 Vers le concept de style d'évolution

Nous avons puisé notre inspiration dans la notion de style architectural que l'on trouve dans le domaine des architectures logicielles. Notre objectif n'est pas d'encapsuler de l'expertise pour la conception, mais plutôt de l'expertise pour l'évolution. C'est ainsi que nous proposons le concept de *style d'évolution*. Le terme "style" véhicule une charge sémantique forte et nous discutons ci-après de la légitimité d'une telle notion pour aborder le problème de l'évolution architecturale.

3.1.1 Thésaurisation, extraction et représentation de l'évolution

Notre recherche vise à thésauriser les expertises pour faciliter l'évolution dans les architectures logicielles à travers leur réutilisation. Pour atteindre cet objectif, il est nécessaire d'extraire l'évolution pour pouvoir la représenter dans un formalisme donné. La thésaurisation, l'extraction ainsi que la représentation de l'évolution sont les pierres angulaires de la réutilisation de l'évolution.

3.1.1.1 Thésaurisation de l'évolution

L'idée de se constituer ses propres bibliothèques d'éléments réutilisables n'est pas nouvelle dans le monde des produits. En revanche, elle l'est beaucoup moins dans le monde des processus, et tout simplement inexistante dans le domaine de l'évolution logicielle. Dans ce dernier cas, l'objectif est de thésauriser les savoir et savoir-faire des personnes ayant déjà fait face à des évolutions, en vue de les réutiliser dans des situations similaires. Pour une entreprise de développement logiciel, disposer d'une bibliothèque d'évolutions déjà validées améliore la qualité du résultat et donne un avantage concurrentiel certain. Néanmoins, cette formule exige de mettre en place une classification claire et évolutive pour éviter les doublons et éviter le temps perdu en recherches infructueuses. En somme, l'efficacité de la réutilisation dépendra aussi de l'infrastructure de réutilisation proposée.

3.1.1.2 Extraction de l'évolution

Nous considérons que tout élément architectural possède sa propre structure, ses comportements mais aussi ses possibilités d'évolution dans l'avenir. Or, nous observons que la composante évolution est traditionnellement amalgamée dans la composante comportement, entravant de ce fait la possibilité de la réutiliser. A la place, nous suggérons de décliner le comportement en deux types : le comportement stable (*steady-state behavior*) qui décrit des services rendus et le comportement d'évolution (*evolution behavior*) qui porte sur la structure et/ou sur le comportement stable. Nous rappelons que nous nous intéressons ici à l'évolution de la structure. Comme l'illustre la Figure 3.1, nous prônons l'extraction de l'évolution en tant que composante à part entière d'un élément architectural.

En procédant de la sorte, il devient possible d'analyser et de modéliser l'évolution d'un système à l'instar de ce qui est entrepris pour le système lui-même. Par conséquent, nous suggérons d'intégrer la subdivision suivante dans un modèle de développement :

– Analyse & conception générale du système : délimiter le problème à résoudre en énumérant un ensemble d'exigences. Puis, déterminer les composants ou les modules nécessaires pour créer le domaine des applications. Cette tâche est confiée à un ou plusieurs architectes d'applications.

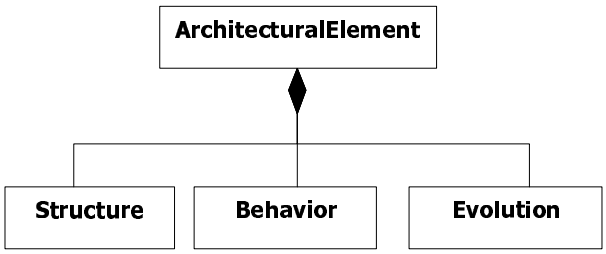

Figure 3.1 – Les trois composantes d'un élément architectural : structure, comportement et évolution.

– Analyse & conception de l'évolution du système : déterminer l'éventail des évolutions possibles de chaque composant. Il s'agit de préparer le système à s'adapter à des situations attendues ou inattendues par le biais de moyens prévus. Cette tâche est confiée à un ou plusieurs architectes d'applications évolutives.

3.1.1.3 Représentation de l'évolution

La structure proposée doit être vue comme un réceptacle de connaissances, de compétences, propre à prendre à son compte certains choix et certaines tâches d'évolution. C'est ainsi que la notion de style d'évolution émerge et que nous suggérons de représenter par les éléments suivants :
– *son type* : la définition abstraite du style,
– *son implémentation* : la mise en oeuvre du style,
– *son instance* : une évolution particulière est un processus exécutable, instance d'un type de style.

La spécification d'un style d'évolution repose d'abord et essentiellement sur son type. En effet, nous n'examinons pas une par une toutes les évolutions concrètes de l'architecture mais travaillons directement sur les archétypes d'évolution qui tiennent lieu de toutes les évolutions possibles. Nous raisonnons ainsi sur des intentions et non des extensions. Dans le reste de ce manuscrit, si aucune précision n'est donnée, le terme "style d'évolution" fera implicitement référence à son type.

3.1.2 Points de vue et styles pour l'évolution d'architectures

Il y a plusieurs vues, tout comme il y a plusieurs structures, chacune avec ses propres objectifs et ses propres orientations dans la compréhension du système. Ainsi, une description architecturale sélectionne un ou plusieurs point de vue. Ce choix dépend des préoccupations des intervenants. Nous donnons une idée de la relation entre les points de vues et les styles dans une architecture par la Figure 3.2.

Nous rappelons qu'une *vue* est une représentation d'une architecture dans la perspective d'un ensemble de préoccupations connexes (IEEE Std 1471, [IEE00]). Cette représentation comprend un ensemble d'éléments du système et les relations qui leur sont associées. Les types d'éléments et les relations ainsi que d'autres méta-informations dans les vues sont décrites par les *points de vue* (IEEE Std 1471, [IEE00]) afin de documenter et de communiquer sur les vues sans ambiguïtés. Par conséquent une vue est une instance d'un point de vue pour un système particulier car

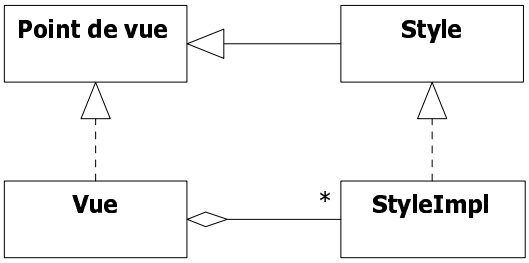

Figure 3.2 – Vues, points de vue, et styles (inspiré de [AZ05])

les éléments et les relations contenus dans la vue sont des instances des types d'éléments et de relations correspondant contenus dans le point de vue. Un *style*, d'un autre coté, définit également des types d'éléments et de relations qui travaillent ensemble afin de résoudre une classe de problèmes dans une perspective. Un style peut être considéré comme une spécialisation d'un point de vue puisque il offre des sémantiques spécifiques aux types d'éléments et de relation et détermine leur utilisation.

La vue structurelle et la vue comportementale sont des exemples de vues importantes d'une architecture logicielle. La première décrit comment un système est structurellement décomposé en composants et connecteurs. La seconde décrit le comportement des composants et des connecteurs et la façon dont ils interagissent ensemble. Dans la vue structurelle, nous pouvons appliquer les styles structuraux *blackboard* ou *pipe&filter* par exemple. Dans la vue comportementale, nous pouvons appliquer les styles comportementaux *broadcast* ou *peer-to-peer* par exemple [1]. A notre connaissance, la vue évolutive de l'architecture n'a jamais été considérée de façon explicite. En toute logique, nous suggérons appliquer dans cette vue des styles d'évolution. Nous proposons de définir un style d'évolution comme suit :

> *Un style d'évolution capture une manière caractéristique de procéder à l'évolution de tout ou partie d'une architecture logicielle. Il sert de guide pour un architecte qui doit alors se conformer au style.*

Cette définition éclaire notre volonté à traiter le problème de l'évolution architecturale par les styles. Dans la section suivante, nous proposons un nouveau modèle d'évolution, baptisé SAEM, reposant sur la notion de style d'évolution.

3.2 SAEM : Style-based Architectural Evolution Model

Nous proposons un nouveau modèle d'évolution à base de styles (*Style-based Architectural Evolution Model – SAEM*). Dans ce modèle, une évolution est réifiée comme un style et attachée à chaque élément architectural, en tant que partie intégrante de sa description tout en étant distincte. Après avoir discuté des bonnes pratiques pour la genèse d'un nouveau modèle d'évolution, nous présentons SAEM à travers son méta-modèle en ayant pris soin d'expliquer ce choix.

1. En ce sens, la notion de style architectural étudié dans la littérature incarne très souvent à la fois un style structurel et un style comportemental.

3.2.1 Propriétés attendues de SAEM

Définir un modèle d'évolution est une partie importante de notre travail. Mais il est tout aussi important d'assurer l'efficacité d'un tel modèle. Pour cela, nous y avons associé trois propriétés importantes :

- La minimalité : l'idée basique derrière cette propriété est de fournir un ensemble minimal et suffisant de concepts dans le modèle d'évolution, ainsi appelé "noyau". En règle générale, un nombre réduit de concepts raccourcit la courbe d'apprentissage.
- La complétude : étroitement liée à la minimalité, la complétude permet d'étendre le noyau sans perturber la façon dont il fonctionne. C'est une sorte de principe d'ouverture/fermeture [2] appliqué à un modèle d'évolution.
- La modularité : l'évolution doit venir compléter l'existant. De cette façon elle n'est pas intrusive, elle est pleinement réutilisable et l'emphase de l'architecte peut être portée uniquement sur l'évolution.

En fait, l'adhésion d'un modèle d'évolution à ces propriétés dépend de ses fondations conceptuelles. Les fondations conceptuelles du SAEM sont décrites par un méta-modèle.

3.2.2 Pourquoi un méta-modèle ?

Nous visons à définir un modèle générique. Cela passe par la méta-modélisation. En effet, la définition d'un méta-modèle permet de ne pas limiter notre approche à un simple langage ou une notation de modélisation. L'approche se place à un niveau d'abstraction supérieur aux langages et notations de modélisation. Nous nous intéressons principalement aux concepts utilisés pour exprimer l'évolution dans les architectures logicielles. Il s'agit de réifier ces concepts et de représenter explicitement leurs caractéristiques et les relations qui existent entre eux. Ainsi, l'approche s'abstrait de toute syntaxe et manipule les éléments de modélisation comme des entités sémantiques et non des entités syntaxiques. Il suffit d'associer aux dites entités sémantiques, soit des entités syntaxiques pour constituer un langage de description, soit des entités graphiques pour constituer une notation graphique de modélisation.

3.2.3 Le méta-modèle SAEM

Les concepts de base de notre méta-modèle SAEM sont décrits à travers la Figure 3.3. Nous nous appuyons sur une modélisation objet pour la description du méta-modèle. Nous utiliserons précisément le formalisme du diagramme de classes de la notation UML 2.0. Ainsi, les concepts clés de classe, d'héritage, de composition et d'association sont exploités, ainsi que les concepts de rôles et de multiplicités sur les associations.

Le diagramme de classe met en évidence les concepts proposés par SAEM, ainsi que les associations qui existent entre eux. Il se focalise principalement sur la manière dont sont organisés les concepts. Nous décrivons en détail ces concepts dans la section suivante.

2. En programmation objet, le principe d'ouverture/fermeture stipule que les entités logicielles doivent être ouvertes à leur extension, mais fermées à leur modification.

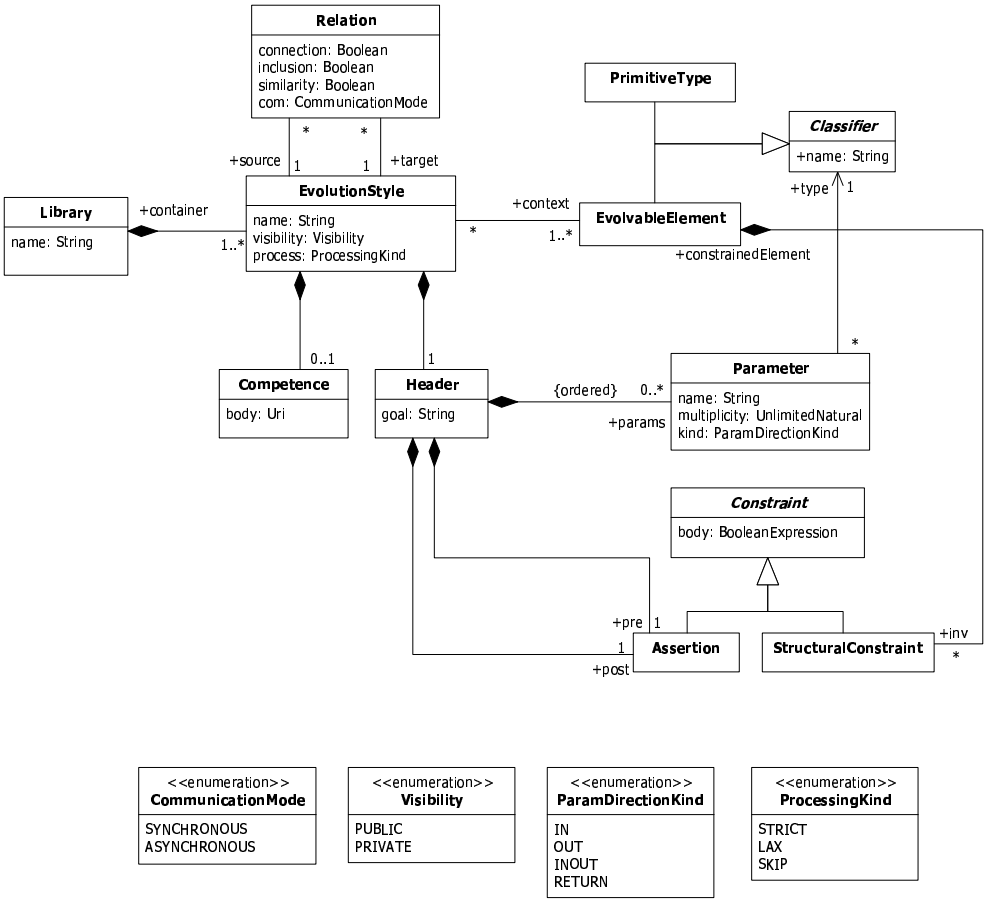

Figure 3.3 – La structure du méta-modèle noyau de SAEM.

3.3 Description des concepts de SAEM

Pour chaque concept du méta-modèle SAEM, nous apportons une définition ainsi qu'un exemple illustratif quand cela est nécessaire.

3.3.1 Élément évolutif

3.3.1.1 Définition

La classe *EvolvableElement* représente une abstraction forte qui peut recouvrir n'importe quel élément architectural dont les instances sont susceptibles d'évoluer. Nous avons introduit ce concept dans SAEM pour modéliser et réifier tout élément significatif d'une architecture évolutive. Si un concept architectural est instance de cette classe (au sens orienté objet), alors il devient possible de lui associer des styles d'évolution. Aussi, cette classe particulière est le point de jonction entre le méta-modèle SAEM et tout concept architectural réifié.

3.3.1.2 Exemple

Il est nécessaire d'identifier quels éléments architecturaux sont susceptibles d'évoluer. Plus le nombre d'éléments réifiés par un ADL est important, plus les possibilités d'évolution sont vastes. Pour comprendre, reconsidérons l'exemple illustratif donné en fin de chapitre 1 dans la section 1.3.4.

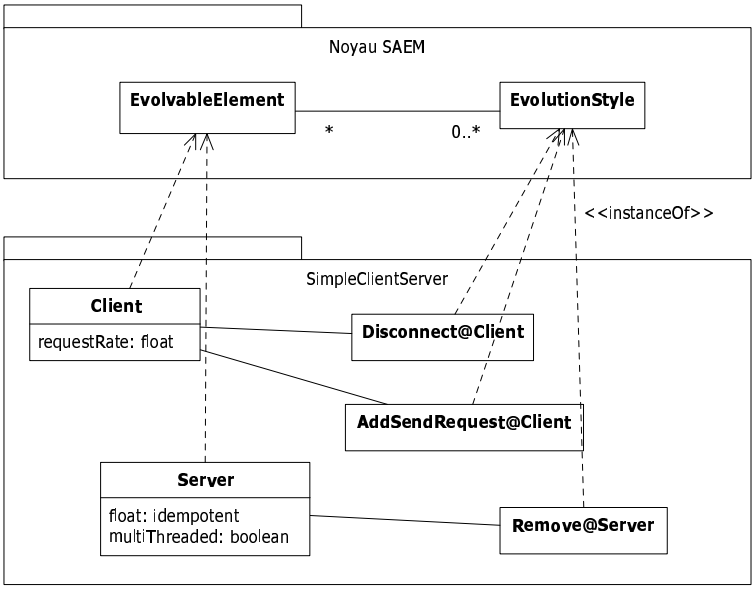

Figure 3.4 – Identification des éléments architecturaux susceptibles d'évoluer.

A travers la Figure 3.4, le choix est fait de considérer les types de composants *Client* et *Server* comme des instances de *EvolvableElement*. En attachant des styles d'évolution aux éléments du niveau A1 d'une architecture logicielle client/serveur, l'évolution pourra être gérée au niveau A0. Dans notre exemple, le client peut évoluer selon les styles *Disconnect@Client* et *AddSendRequest@Client*, tandis que le serveur peut évoluer selon le style *Remove@Server*. D'autres styles peuvent bien évidemment être associés à ces éléments architecturaux.

3.3.2 Style d'évolution

3.3.2.1 Définition

Un style d'évolution encapsule ce qui permet de décrire et d'appliquer une évolution à un élément architectural. La classe *EvolutionStyle* est une entité nommée qui est composée de deux parties complémentaires : un entête et une compétence. L'entête est obligatoire tandis que la compétence ne l'est pas. Quand la compétence n'est pas spécifiée, on dit dans ce cas que le style d'évolution est *abstrait*. Examinons plus en détail ce que sont l'entête et la compétence :

- L'entête possède une description informelle du but poursuivi et publie une liste de paramètres et d'assertions. L'élément évolutif apparaît comme un paramètre implicite nommé `context`. Le type des paramètres sont fournis par l'ensemble des éléments évolutifs, plus les types primitifs usuels (String, int, boolean, float, etc.)

– La compétence décrit une unité d'implémentation correspondant à l'entête. L'unité d'implémentation spécifie le flot des données et toute la logique de contrôle. L'implémentation est considérée comme une ressource externe, identifiable par son URI (*Uniform Resource Identifier*).

Il est important de remarquer que nous dissocions un style d'évolution de l'élément architectural concerné. L'objectif est que l'élément soit lié à son ou ses styles d'évolution. C'est une façon d'ajouter la composante évolution manquante dans la description des éléments architecturaux, comme cela a été défendu au début de ce chapitre (cf. section 3.1.1).

3.3.2.2 Exemple

La Figure 3.5 schématise la représentation d'un style selon trois compartiments : le nom du style, son entête et sa compétence. La convention de nommage utilisée dans le reste de ce manuscrit suit le motif *Evolution@ElementEvolutif*.

Le style *RemoveTail@Component* décrit la suppression d'un composant qui se trouve à l'extrémité de sa configuration d'appartenance (`self.context.owner`). Ceci est exprimé par la précondition `self.context.owner.bindings->exists(b|b.target=context)`, qui indique que le composant est la cible d'au moins un binding de la configuration. Le compartiment de la compétence de ce style indique que pour la suppression d'un tel composant, il faut, dans l'ordre :

– Supprimer les bindings auquel le composant participe.
– Supprimer toutes les interfaces du composant.
– Exclure le composant de la liste des composants de la configuration.
– Détruire le composant proprement dit, par désallocation de celui-ci.

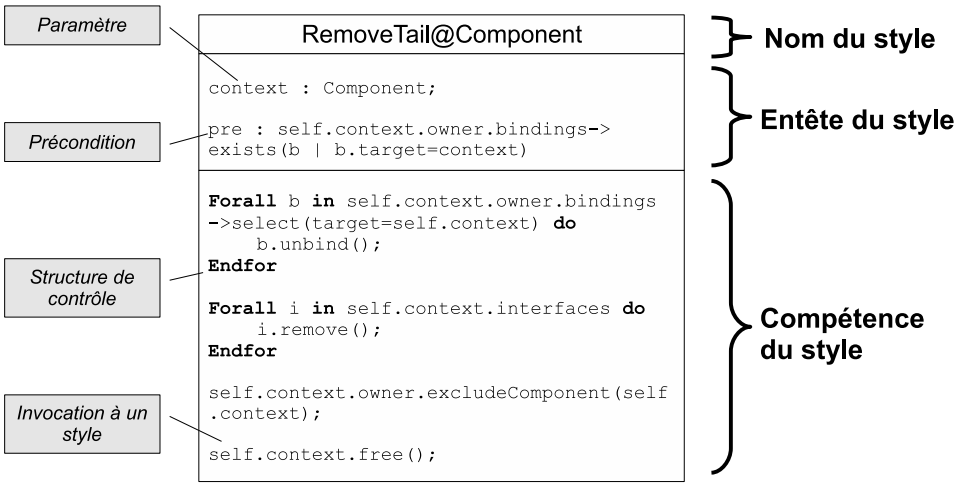

Figure 3.5 – Nom, entête et compétence d'un style d'évolution.

Dans SAEM, toute opération d'évolution est vue comme un style. Le style d'évolution est en relation avec d'autres styles (non montré sur la figure), pour lui permettre de les invoquer. Dans l'exemple, on peut deviner les styles d'évolution *Unbind@Binding*, *Remove@Port*, *Exclude-Component@Configuration* et enfin *Free@Component*. On considère ici que l'invocation suit une

orientation objet de la forme *ElementEvolutif.Evolution([paramètres éventuels])*. Sur ce principe, un architecte ou un autre style d'évolution pourra invoquer le style *RemoveTail@Component* sur un composant `c1` à travers l'instruction *c1.RemoveTail()*.

3.3.3 Contrainte

3.3.3.1 Définition

Les contraintes ont pour objectif principal de permettre de contrôler l'évolution des éléments d'une architecture. La classe abstraite *Constraint* est dérivée en deux sous-classes :

- Contrainte Structurelle : représente une contrainte sur la structure d'un élément évolutif. Elle doit être respectée tout au long du cycle de vie de l'élément.
- Assertion : représente des contraintes sur l'état de tout ou partie de l'architecture avant, pendant, et après son évolution.

La spécification d'un style d'évolution inclut une précondition et une postcondition dans son entête. Tandis que la précondition permet de vérifier que l'élément évolutif est dans le bon état pour pouvoir y appliquer le style d'évolution, la postcondition informe de l'état dans lequel ce dernier sera après y avoir appliqué le style d'évolution. L'entête ne définit pas explicitement d'invariant : celui-ci provient directement de la contrainte structurelle de l'élément évolutif concerné. En effet, la contrainte structurelle posée sur un élément évolutif constitue naturellement un invariant vis-à-vis de son évolution. L'ensemble de ces éléments permet d'établir un contrat lors de la réutilisation d'un style d'évolution.

3.3.3.2 Exemple

Pour ne pas présupposer d'un langage de contrainte en particulier, nous considérerons une syntaxe "à la OCL ". Par conséquent, nous disposons d'un langage simple, supportant des expressions de navigation et des expressions booléennes (conjonction, disjonction, implication, quantification, etc.). Pour pouvoir exprimer plus de choses on y adjoint un moyen pour que les postconditions puissent se référer à l'ancienne valeur des éléments modifiés par la compétence (mots clé `@pre` et `@post`).

Nous donnons ci-dessous un exemple de contrainte structurelle associée à l'élément évolutif *Client*. Cette dernière s'assure qu'un composant de type client ne possède pas plus de quatre ports de type *sendRequest*.

```
context Client
  inv : self.ports->select(isTypeOf(sendRequest))->size()<=4
```

Nous donnons ci-dessous un exemple d'assertions associées au style d'évolution *AddSendRequest@Client*. La précondition vérifie que le composant de type client possède un taux de requête strictement supérieur à la valeur 10. La postcondition vérifie que le composant possède bien un port de plus qu'avant son évolution.

```
context AddSendRequest@Client::invoke():void
  pre : self.context.requestRate > 10
  post : self.context@post.ports->size() = self.context@pre.ports->size()+1
```

Invariablement, le style d'évolution est tenu de respecter la contrainte structurelle posée sur le type de composant client. Ceci est particulièrement important dans notre exemple où il y un risque d'incohérence structurelle puisque le style *AddSendRequest@Client* pourrait faire passer le nombre de port à 5.

3.3.4 Relation

3.3.4.1 Définition

Dans la continuité des travaux de Roger Chaffin [Cha88], nous définissons le concept de relation entre deux styles d'évolution de façon générique par un triplet d'éléments relationnels, auquel nous associons un mode de communication. Selon cette stratégie, la présence ou l'absence des éléments relationnels (a) de connexion, (b) d'inclusion et (c) de similarité permet de déterminer la nature de la sémantique portée par une relation entre deux concepts. En procédant ainsi, il devient aisé de se doter de relations sémantiques adaptées à ses besoins, sans devoir apporter de modification au méta-modèle de SAEM. En outre, chaque relation est dirigée et supporte une communication par message entre le style « émetteur » et le style « receveur » dont le mode peut être synchrone ou asynchrone :

- Synchrone : la réception d'un message est "calé" sur le moment de son émission. Ici, ce processus est bloquant, c'est-à-dire que le style d'évolution émetteur doit attendre la terminaison du style receveur pour pouvoir continuer son exécution.
- Asynchrone : la réception d'un message peut être postérieure au moment de son émission. Ici, ce processus est non-bloquant, c'est-à-dire que le style d'évolution émetteur peut continuer son exécution sans même attendre la terminaison du style receveur.

3.3.4.2 Exemple

Nous instancions le concept *Relation* afin d'obtenir trois relations sémantiques que nous jugeons essentielles pour notre approche : l'*utilisation* (ou référence), la *composition* et la *spécialisation*. Le diagramme d'objet UML de la Figure 3.6 montre comment ces relations sont instanciées depuis leur classe commune définie dans le méta-modèle de SAEM.

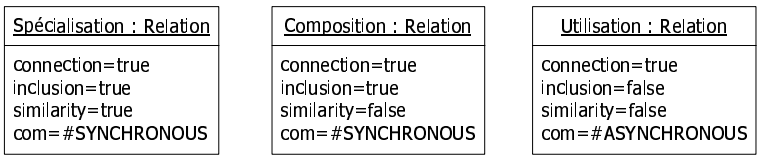

Figure 3.6 – Instances du concept *Relation* de SAEM : spécialisation, composition et utilisation.

Ainsi, à l'instar des relations inter-styles architecturaux présentées en Section 1.1.4.3, le modèle SAEM prévoit trois relations inter-styles d'évolution. La spécialisation sera utilisée pour créer des styles d'évolution plus spécifiques et mieux adaptés à certaines classes de problèmes d'évolution. La composition sera utilisée pour créer des styles composites, par combinaison de styles d'évolution existants. Enfin, l'utilisation sera utilisée pour faire collaborer des styles d'évolution. La table 3.1 présente des informations complémentaires sur ces trois types de relations.

En premier lieu, on remarque que chacune des trois relations possède au minimum l'élément de connexion, qui est la quintessence du rôle d'une relation. Ensuite, l'élément d'inclusion fait

Nom	*Signification*	*Éléments relationnels*	R	T	S
Spécialisation	« est-une-sorte-de »	{Connexion, Similarité, Inclusion}	√	√	–
Composition	« est-composé-de »	{Connexion, Inclusion}	√	√	–
Utilisation	« utilise »	{Connexion}	–	–	–

Table 3.1 – Éléments relationnels et axiomes mathématiques de réflexivité (R), de transitivité (T) et de symétrie (S) associés aux trois relations sémantiques.

émerger des niveaux hiérarchiques entre les styles d'évolution, caractéristique de la relation de composition et de la relation de spécialisation. Cette caractéristique est confirmée par la relation d'ordre que ces deux relations définissent axiomatiquement : elles sont toutes deux réflexives, antisymétriques et transitives. Enfin, la spécialisation se distingue par la présence de l'élément de similarité, qui traduit une continuité conceptuelle entre les styles reliés. La présence ou l'absence de ces trois éléments relationnels dans le triplet utilisé pour définir des relations sémantiques suggère qu'il existe un ordre entre ces dernières. Ainsi, on considère que plus (respectivement moins) une relation peut être décrite par des éléments relationnels, plus elle offre une sémantique riche (respectivement faible). Avec les trois relations essentielles de SAEM nous obtenons l'ordre suivant :

SPÉCIALISATION > COMPOSITION > UTILISATION

3.3.5 Bibliothèque

Une bibliothèque (classe *Library* dans la Figure 3.3) est une unité structurante destinée à contenir un ensemble de styles d'évolution. Le concept de bibliothèque est similaire au concept d'API (*Application Programming Interface*) dans l'univers de la programmation. En ce sens, une bibliothèque expose une interface, autorisant la réutilisation de tout ou partie de son contenu. Une bibliothèque fournit un espace de nom spécifique et contrôle son interface vis-à-vis de l'extérieur en jouant sur la visibilité des styles d'évolution, qui peuvent être privés ou publiques. Ainsi, il est possible de spécifier des styles d'évolution uniquement visibles et donc utilisables depuis l'intérieur de la bibliothèque. Par ailleurs, la bibliothèque constitue le point d'entrée unique pour l'accès aux styles d'évolution et est considéré comme un élément de modélisation racine. Enfin, le concept de bibliothèque offre un moyen de distribution bien défini des styles d'évolution. De ce point de vue, les librairies peuvent être entreposées, échangées, copiées, enrichies, interrogées, etc. Des politiques relatives à leur distribution peuvent même leurs être associées.

3.4 Mécanique opératoire de SAEM

Dans cette section, nous revenons sur les différentes relations de SAEM évoquées depuis le début de ce chapitre, en expliquant la mécanique opératoire associée à chacune d'elles. Comme cela est schématisé par la Figure 3.7, les mécanismes opérationnels fournis par SAEM sont l'instanciation, la spécialisation, la composition, et enfin l'utilisation. Ces mécanismes s'inspirant largement de ceux de l'orienté objet, nous pouvons de ce fait reprendre à notre compte les notations du diagramme de classe UML pour les illustrations à suivre dans cette section.

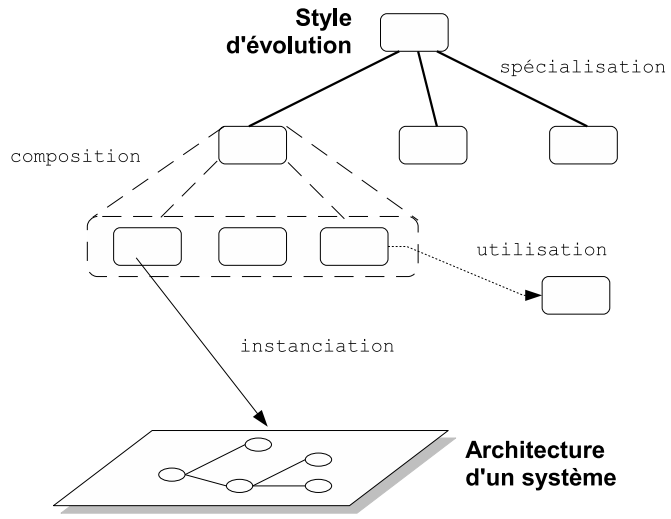

Figure 3.7 – Les styles d'évolution sont reliés entre eux par la spécialisation, la composition et l'utilisation. Ils sont instanciés sur l'architecture d'un système.

3.4.1 L'instanciation

D'une manière générale, l'instanciation est un mécanisme qui permet de passer d'un niveau de modélisation donné au niveau inférieur. Les styles d'évolution peuvent être instanciés plusieurs fois dans une architecture. L'instance d'un style d'évolution est un processus particulier qui est créé dans le respect de la structure donnée par son style. Toutes les instances d'un style d'évolution doivent offrir exactement la même compétence que ce dernier. Le mécanisme d'instanciation recouvre la liaison des paramètres formels à des paramètres effectifs (*i.e.*, des éléments d'une architecture), l'évaluation de la précondition et de la postcondition, et l'exécution du corps de la compétence. Par conséquent, un style d'évolution abstrait ne peut pas avoir d'instances étant donné qu'il ne spécifie aucune compétence. On peut considérer qu'une instance est rattachée à son type de style d'évolution par la relation "est-un" (*is-a*).

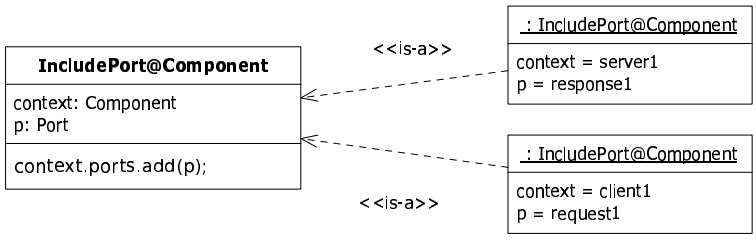

Figure 3.8 – Mécanisme d'instanciation de style d'évolution.

L'exemple de la Figure 3.8 montre deux instanciations du même style d'évolution *IncludePort@Component* sur des éléments particuliers d'une architecture logicielle. Dans le premier cas, l'évolution vise l'ajout du port `response1` à la liste des ports du composant `server1` ; dans le second cas, l'évolution vise l'ajout du port `request1` à la liste des ports du composant `client1`.

3.4.2 La spécialisation

Les styles d'évolution peuvent être définis par extension d'autres styles. Le mécanisme d'héritage associé à la relation de spécialisation est inspiré du mécanisme d'héritage des classes dans le paradigme objet. Avec la structure bipartite proposée, d'une part un sous-style peut ajouter et surcharger des éléments de l'entête de son super-style, et d'autre part un sous-style peut redéfinir la compétence de son super-style. Ce mécanisme peut être utilisé pour définir des styles d'évolution concrets en tant que sous-styles de styles abstraits en fournissant la compétence manquante. De façon plus détaillée, nous avons :

- Surcharge d'entête (*Header overloading*) : les paramètres (incluant `context`), les assertions et les relations de composition et d'utilisation décrites dans le super-style sont autant d'éléments hérités par le sous-style. L'architecte est alors libre d'ajouter de nouveaux paramètres et de nouvelles relations. Au besoin, l'architecte peut spécialiser les paramètres et les relations existantes, renforcer la précondition et affaiblir la postcondition.
- Redéfinition de compétence (*Competence overriding*) : la compétence définie dans le super-style est héritée dans le sous-style. L'architecte est alors libre de la remplacer par une nouvelle implémentation. Lors de la redéfinition, on laisse la possibilité à l'architecte de réutiliser la compétence du super-style si ce dernier n'est pas abstrait. Le cas échéant, la communication entre le sous-style et le super-style est synchrone.

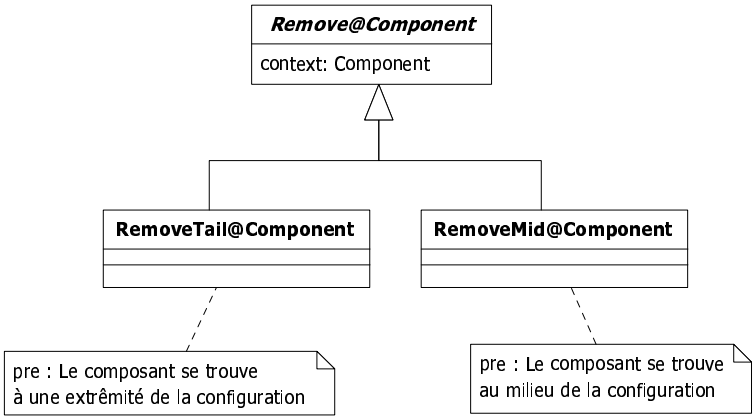

Figure 3.9 – Mécanisme de spécialisation de style d'évolution.

L'héritage proposé sera utilisé pour créer des hiérarchies conceptuelles spécialisées. Les styles abstraits pourront être utilisés comme un moyen simple pour factoriser un ensemble de styles partageant une même stratégie d'évolution. Pour ne pas introduire de conflits sémantiques entre sous-styles, SAEM ne propose que l'héritage simple.

L'exemple de la Figure 3.9 montre la spécification de deux sous-styles, par spécialisation du style abstrait *Remove@Component*. Chaque sous-style fournit la compétence manquante, compte tenu de la situation du composant avant évolution, déterminée par une précondition particulière. Dans cet exemple, la surcharge d'entête n'est pas exploitée.

3.4.3 La composition

La composition est nécessaire pour décrire les évolutions à différents niveaux de détails. La composition de style se réfère à la structuration « tout-partie » entre deux styles. A chaque niveau de composition, chaque style peut être vu comme ayant pour parties ces sous-styles qui représentent des étapes entrant dans sa compétence. Cette relation suggère un fort couplage entre les styles d'évolution et favorise l'encapsulation des compétences complexes. Le SAEM utilise le mécanisme de composition pour définir des styles d'évolution composites, de plus en plus complexes, déléguant leurs fonctionnalités aux styles composants. La composition démarre d'un ensemble de styles d'évolution primitifs, dont la compétence est élémentaire. Enfin, le mode de communication attribué à la composition est nécessairement synchrone, car l'exécution des styles composants est subordonnée à celle du style composite.

Figure 3.10 – Mécanisme de composition de style d'évolution.

L'exemple de la Figure 3.10 montre la spécification du style d'évolution *Remove@Component* composé du style *Remove@Port*. Cette relation est le reflet du lien de composition entre les éléments évolutifs eux-mêmes, et s'avère être un moyen d'exprimer que la suppression du composant ne va pas sans la suppression du ou des ports qui le composent.

3.4.4 L'utilisation

Dans SAEM, l'utilisation est une technique fondamentale dans la perception d'un système « expert » comme un ensemble d'expertises inter reliées. Ainsi, l'utilisation est une relation permettant à un style de référencer un autre style pour en utiliser la fonctionnalité. Elle ne doit pas être confondue avec une relation de composition, car elle a un caractère plus momentané et n'implique pas de couplage fort. Enfin, le mode de communication attribué à l'utilisation est asynchrone, car les exécutions des styles n'ont pas nécessairement besoin d'être concordantes.

Figure 3.11 – Mécanisme d'utilisation de style d'évolution.

L'exemple de la Figure 3.11 montre la spécification du style d'évolution *Move@Port* utilisant la spécification du style *Rename@Port*. Cela s'explique par le fait que le déplacement d'un port d'un composant à un autre peut nécessiter son renommage pour ne pas introduire de conflits avec les ports existants dans le composant destination.

3.5 Formalisme de représentation des styles d'évolution

La spécification des styles d'évolution doit être aussi simple que possible, tant sur le plan des concepts manipulés que sur celui de la quantité d'éléments à décrire. Compte tenu des fonctionnalités offertes par SAEM, le formalisme choisi doit permettre de distinguer l'entête de la compétence d'un style d'évolution ainsi que les éléments évolutifs concernés. En outre, l'architecte doit pouvoir hiérarchiser les entêtes, les compétences et les éléments évolutifs et disposer de plusieurs vues sur ses styles. Le modèle en Y (MY) est un formalisme qui intègre naturellement ces fonctionnalités et qui a en outre été utilisé par notre équipe pour décrire les architectures logicielles à base de composants [SOK08]. En utilisant MY comme support, un style d'évolution peut être décrit par trois aspects (correspondant aux trois branches du Y) : domaine, entête et compétence. Ces aspects représentent successivement tout ce qui est lié aux éléments évolutifs, aux opérations d'évolutions et aux implémentations de ces opérations.

3.5.1 MY : La méta-modélisation en Y

La modélisation, la méta modélisation et les bibliothèques de composants réutilisables sont des techniques éprouvées qui sont utilisées dans l'ingénierie de la connaissance (*Knowledge Engineering*, KE). La modélisation et la méta modélisation sont des techniques utilisées pour représenter des systèmes à un niveau élevé qui abstrait les considérations d'implémentation et se concentre sur la compétence [Men02]. Dans le domaine de la représentation de connaissances, on parle de la méta-connaissance pour évoquer la connaissance concernant une connaissance.

Les bibliothèques de composants réutilisables ont été décrites dans différentes approches de KE [Mot00]. Après avoir examiné ces bibliothèques, nous avons constaté qu'elles font face à des problèmes très proches des nôtres. En ce qui nous concerne : comment décrire un style d'évolution, ce que devrait être le niveau de granularité d'un style évolution, comment structurer et classer les style d'évolutions de sorte que l'utilisateur puissent facilement comprendre, trouver et utiliser des styles afin de transformer des architectures.

Les approches classiques de KE telles que CommonKADS [Bre94], TINA [Ben95], VITAL [SMR93], MIKE [AFLS98], Protégé [GMF*03], les tâches génériques [CJS92], les composants de l'expertise [Ste90], et MY [OMK02] décrivent des systèmes à base de connaissance (SBC) en utilisant trois éléments complémentaires : tâche, méthode de résolution de problème (*Problem Solving Methods*, PSMs) et domaine. La séparation des méthodes (utilisées pour réaliser une tâche donnée) de la tâche et du domaine permet de définir un SBC comme une combinaison de ces trois éléments. L'approche MY combine la méta modélisation et les bibliothèques de composants réutilisables.

3.5.2 Style d'évolution en Y

Comme cela vient d'être évoqué, en utilisant le modèle MY, un style d'évolution peut être décrit par trois aspects : domaine, entête, compétence.

- le domaine représente la connaissance disponible/les éléments évolutifs,
- l'entête représente les opérations réalisables sur la base de cette connaissance,
- la compétence représente la méthode choisie pour réaliser l'opération.

Ces trois concepts sont reliés entre eux pour établir un style d'évolution. Ils sont définis dans un but de réutilisation pour décrire différents archétypes d'évolution. Par conséquent, nous pouvons réutiliser un MY pour décrire différents styles d'évolution dans différents contextes :

- un domaine peut être utilisé pour décrire les systèmes client/serveur, pipe&filter, etc.,
- un entête peut être utilisé pour décrire la deconnexion d'un client, le retrait d'un serveur, l'ajout d'un filtre, etc.,
- une compétence peut être utilisée pour décrire le retrait d'un serveur avec transfert de ses clients ou non, etc.

3.5.2.1 Les concepts de base

Nous l'avons vu, les deux concepts principaux de la description d'un style d'évolution sont l'entête et la compétence. Il existe en réalité un troisième concept, non explicité jusqu'ici, qui est le domaine. Le domaine offre un vocabulaire composé des éléments évolutifs et de leurs relations, typiquement sous formes de classes et d'attributs dans les représentations par objets. L'acquisition de ce vocabulaire conceptuel de base est donc nécessaire pour exprimer la connaissance en matière d'évolution sur ce domaine. Nous modélisons ces concepts en utilisant trois aspects (chacun est représenté par une branche de la lettre Y) : aspect domaine, aspect entête, aspect compétence.

Chaque branche de la lettre Y décrit un concept du style d'évolution, comme le montre la Figure 3.12. Le centre de l'architecture en Y décrit un entête primitif lié à une compétence élémentaire et décrits dans la terminologie du domaine. De plus, afin de garder la description des trois concepts (domaine, entête, compétence) indépendants les uns des autres, nous explicitons deux relations : lien inter-concepts et lien intra-concepts. La relation *inter* décrit le lien entre deux concepts différents (par exemple entre un entête et une compétence), alors que la relation *intra* décrit le lien entre deux concepts du même type (par exemple entre un domaine et ses sous-domaines). Ces relations sont utilisées dans le MY comme un outil d'adaptation et d'assemblage pour décrire des styles d'évolution.

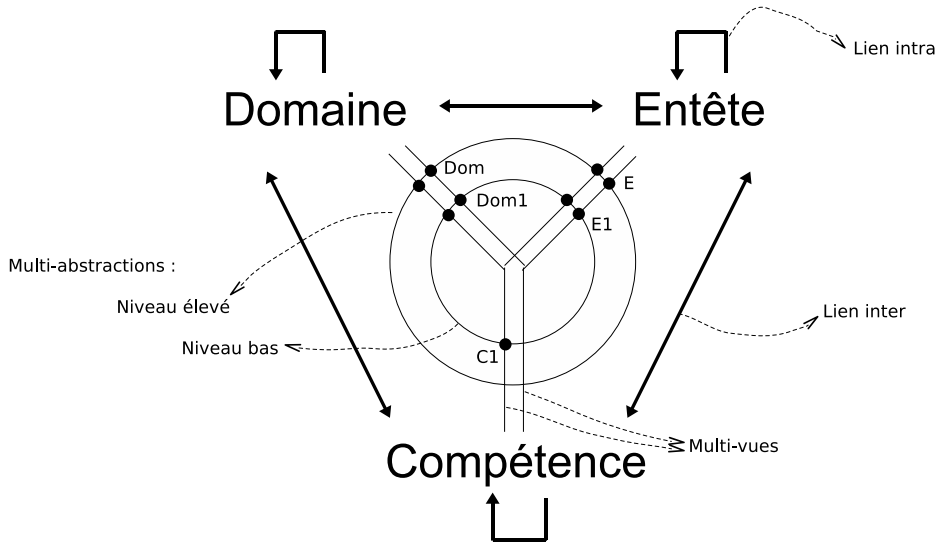

Figure 3.12 – La représentation des concepts du style d'évolution en utilisant le modèle en Y.

La relation intra se décline sous les formes Domaine–Domaine, Entête–Entête et Compétence–Compétence. La spécialisation (est-une-sorte-de), la composition (est-composé-de) et l'utilisation (utilise) définis dans SAEM sont des relations intra-concepts. A titre d'exemple, la spécialisation d'un style d'évolution est vue comme la combinaison d'une spécialisation de son domaine (sous-typage de l'élément évolutif), de son entête (surcharge) et de sa compétence (redéfinition). On remarquera au passage qu'il n'y a pas forcément isomorphisme entre la relation de spécialisation au niveau du style et celle au niveau de ses trois constituants. La même remarque peut être faite pour les relations de composition et d'utilisation.

La relation inter se décline sous les formes Domaine–Entête, Domaine–Compétence et Entête–Compétence. Des relations sémantiques jusqu'alors implicites émergent comme des relations inter-concepts. Il s'agit de la relation d'expression (est-exprimé-sur) et de la relation de réalisation (implémente). La première sert à relier un entête et une compétence à un domaine. La seconde sert à relier une compétence à son entête.

3.5.2.2 Multi-abstractions et multi-vues

Afin de capturer tous les aspects des styles d'évolution et encapsuler leur complexité, MY décrit les styles selon deux découpages : multi-abstractions et multi-vues.

Le découpage multi-abstractions Le découpage multi-abstractions définit différentes abstractions pour un élément donné. Aussi, un style d'évolution peut être décrit en utilisant différents niveaux d'abstraction. Chaque niveau d'abstraction est représenté par un cercle dans le modèle en Y, à partir du niveau le plus élevé (cercle de fort diamètre) au niveau le plus bas (cercle de faible diamètre). Les deux acceptions données aux cercles correspondent à l'abstraction de composition/décomposition et à l'abstraction de spécialisation/généralisation. D'autres acceptions sont possibles si de nouvelles relations sémantiques et hiérarchiques sont introduites dans SAEM. Par exemple, la difficulté à exprimer l'implémentation d'un style d'évolution peut être représentée en utilisant MY et son découpage multi-abstraction via la spécialisation. Dans ce cas, le cercle représentant le niveau le plus élevé d'abstraction montre uniquement un domaine et un entête et nous devons décrire le style à des niveaux d'abstraction inférieurs afin de trouver la compétence. En règle générale, pour obtenir un style d'évolution simplifié avec un minimum d'éléments, nous devons logiquement le décrire à des niveaux d'abstractions plus élevés.

Le découpage multi-vues La découpage multi-vues permet la définition de différentes vues pour les styles d'évolution. Nous présentons trois vues : la vue générique, la vue abstraite et la vue concrète. La vue générique décrit un style d'évolution indépendamment d'un domaine. La vue abstraite décrit un style d'évolution sans fournir sa mise en oeuvre. Enfin, la vue concrète décrit un style d'évolution de manière complète.

Le modèle en Y représente différentes vues en utilisant plusieurs Ys. Chaque Y représente une vue d'un style d'évolution et comment ses différents aspects (domaine, entête et compétence) sont supportés dans cette vue. Par exemple, comme cela est illustré par la Figure 3.13, les domaines sont explicitement représentés dans la vue abstraite et la vue concrète. Cependant, dans la vue générique, les domaines ne sont pas représentés (il faut noter que toutes les vues sont ici illustrées pour un même niveau d'abstraction).

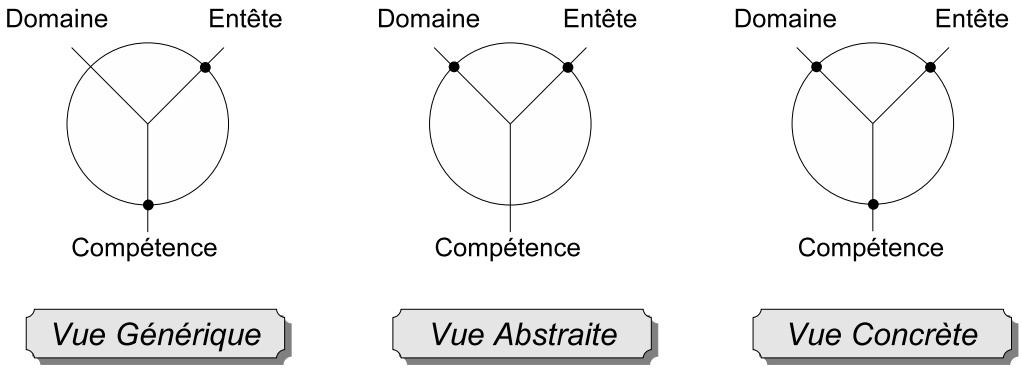

Figure 3.13 – La représentation de vues multiples en utilisant MY.

3.5.3 Exemple d'illustration

Dans cette section, nous traitons l'exemple du style d'évolution *Move@Port*, dont le but est simplement de déplacer un port de son composant d'origine vers un composant de destination (de même type que le composant d'origine). Intuitivement, un déplacement (couper/coller) peut être vu comme la combinaison d'un processus "couper" et d'un processus "coller". Dans les termes de SAEM, ceci signifie que *Move@Port* peut être décrit comme un style composite, incluant les styles d'évolution *ExcludePort@Component* et *IncludePort@Component*. Dans la Figure 3.14, nous utilisons le modèle en Y pour illustrer comment ces styles d'évolution peuvent être représentés. La convention de nommage d'un entête et d'une compétence utilisée dans MY s'inspire d'une bonne pratique issue du monde de la programmation où il est conseillé de préfixer les classes d'interfaces par la lettre `I` et de suffixer les classes d'implémentation par `Impl`.

Dans l'exemple, en considérant l'acception de composition/décomposition, nous avons deux niveaux d'abstraction :

- un niveau d'abstraction élevé dans lequel on trouve le domaine *C&C* (Composant & Connecteur), l'entête *IMovePort* et la compétence *IMovePortImpl*.
- un niveau d'abstraction moins élevé dans lequel on trouve le domaine *C&C*, les entêtes *IExcludePort* et *IIncludePort* ainsi que les compétences *ExcludePortImpl* et *IncludePortImpl*.

Aussi, le modèle MY pour cet exemple ne possède qu'une seule vue : la vue concrète. Dans cette dernière, les trois aspects (domaine, entête et compétence) sont supportés.

3.6 Bilan de SAEM

Nous avons illustré les différents éléments de description du modèle d'évolution SAEM pour spécifier l'évolution d'une architecture logicielle. Nous avons également proposé un formalisme de représentation des styles d'évolution. Dans cette section, nous fournissons les éléments de synthèse relatifs à ce modèle d'évolution à base de style et nous montrons comment ce dernier se positionne vis-à-vis de la pyramide des besoins introduite dans le Chapitre 2 de cette thèse.

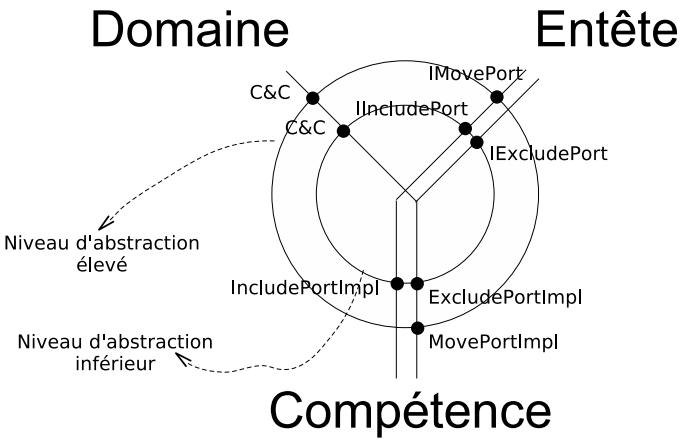

Figure 3.14 – La représentation de deux niveaux d'abstraction pour le style *Move@Port*.

3.6.1 Éléments de synthèse

L'idée clé de notre modèle d'évolution est d'attacher à chaque élément architectural un ensemble de styles d'évolution qui sont autant de possibilités pour son évolution dans le temps. Il est aisé d'augmenter ou de diminuer le nombre de styles disponibles pour un élément architectural. Par essence, le style d'évolution "connaît" l'élément architectural et tient compte de sa structure. Cela est caractéristique d'une spécification de l'évolution a priori, qui se distingue des spécifications a posteriori où aucune garantie n'est donnée sur le choix d'une évolution à appliquer à un élément architectural. Relié à ce dernier point, on remarquera qu'un élément architectural définit un contexte d'exécution pour ses styles d'évolution et qu'il n'est pas nécessaire de redéfinir ces derniers pour chaque sous-élément architectural. En exploitant ainsi le polymorphisme par sous-typage des éléments évolutifs eux-mêmes, l'architecte est amené à découvrir quel est l'ensemble des styles d'évolution disponibles pour chacun des éléments de son architecture. A titre d'exemple, un composant de type *Client* peut évoluer par son style *Disconnect@Client* bien sûr, et intrinsèquement par son style *Remove@Component*, même si ce dernier ne lui est pas explicitement attaché.

Au haut niveau d'abstraction de SAEM, il est important de remarquer que la composition et l'utilisation de styles d'évolution par un autre ne préjuge d'aucune contrainte de précédence dans l'exécution de ces différents styles. Elle n'en précise pas l'ordre d'application et n'a pas vocation à modéliser la dynamique de l'exécution. Elle n'implique pas d'avantage que lors d'une exécution particulière, les différents styles seront systématiquement lancés. De façon à exprimer d'éventuelles contraintes d'enchaînement, il est nécessaire de définir des relations d'ordonnancement telles que par exemple "est précédé de", "se poursuit par", "démarre par", etc.

3.6.2 Positionnement dans la pyramide des besoins

Le tableau 3.2 récapitule le positionnement de SAEM par rapport aux trois niveaux de la pyramide des besoins.

<table>
<tr><th colspan="6">NIVEAU 1 : INTENTION DE SAEM</th></tr>
<tr><th colspan="2">Formuler l'évolution</th><th colspan="2">Gérer les impacts de l'évolution</th><th>Garder la trace de l'évolution</th></tr>
<tr><td colspan="2">√</td><td colspan="2">√</td><td></td></tr>
<tr><th colspan="6">NIVEAU 2 : EXPRESSIVITE DE SAEM</th></tr>
<tr><th>Niveau d'abstraction</th><th>Mode d'expression</th><th>Domaine</th><th colspan="2">Niveau de modélisation</th><th>Mécanique opératoire</th></tr>
<tr><td>- Spécification / Transparence variable
- Code</td><td>- Langage de styles / Granularité variable</td><td>- Structuré en couches (via spécialisation)</td><td colspan="2">- Libre</td><td>Avancée (raisonnement classificatoire)</td></tr>
<tr><th colspan="6">NIVEAU 3 : QUALITE DE SAEM</th></tr>
<tr><th colspan="4">Réutilisabilité</th><th rowspan="2">Support</th><th rowspan="2">Adaptabilité</th></tr>
<tr><th>Extensibilité</th><th>Evolutivité</th><th>Compositionnalité</th><th>Remplaçabilité</th></tr>
<tr><td>√</td><td>√</td><td>√</td><td>√</td><td>Bon</td><td>√</td></tr>
</table>

Table 3.2 – Évaluation de SAEM vis-à-vis de la pyramide des besoins.

Niveau 1 : SAEM est un modèle d'évolution conçu pour formuler l'évolution sous forme de styles et également pour propager les impacts des évolutions à travers les relations sémantiques inter-styles. Nous pensons que la traçabilité de l'évolution est une problématique à part entière et qui peut venir compléter SAEM si besoin, sans en changer le fonctionnement.

Niveau 2 : Les styles sont décrits à un haut-niveau d'abstraction, mais une stratégie pour leur projection vers du code source est proposée dans le Chapitre 4. Par ailleurs, la séparation de l'entête et de la compétence dans la spécification d'un style reconnaît le principe – largement accepté – du masquage de l'information [Par72] et permet de jouer sur le niveau de transparence lors de la réutilisation de l'évolution.

Le SAEM introduit ce que l'on peut appeler un "langage de styles". Un langage de styles est une collection de styles formant un vocabulaire et une démarche pour comprendre et communiquer des idées portant sur l'évolution logicielle. Il décrit la manière avec laquelle les styles sont inter-reliés et la manière avec laquelle ils se complètent dans un tout cohésif. Dans SAEM, toute évolution récurrente constitue un style d'évolution et est ainsi manipulée de manière uniforme, que celle-ci soit à forte ou plus fine granularité.

Il est utile de faire la distinction entre les évolutions qui sont communes à tous les systèmes à base de composants, de celles qui sont communes à une famille particulière de systèmes. Nous désignons la première catégorie sous le terme d'évolution généraliste tandis que nous désignons la deuxième catégorie sous le terme d'évolution dédiée[3]. Nous suggérons que la plupart des évolutions dédiées peuvent être vues comme des extensions des évolutions généralistes afin de les personnaliser à des besoins particuliers. Pour permettre de spécifier des styles d'évolution ayant une portée variable dans SAEM, les couches d'évolution sont obtenues grâce à la relation de spécialisation inter-styles et son mécanisme d'héritage.

L'évolution peut être gérée à n'importe quel niveau de modélisation d'une architecture logicielle ; tout dépend de la façon dont SAEM est utilisé. Dans SAEM, nous nous basons sur le principe suivant : pour pouvoir gérer l'évolution d'une architecture logicielle décrite sur un niveau de modélisation A_i, il faut d'abord identifier le niveau de modélisation supérieur A_{i+1} qui a permis l'instanciation de l'architecture. Tout style d'évolution sera alors spécifié sur la base

3. Les termes anglais correspondants seraient *general-purpose evolution* et *domain-specific evolution*.

des éléments appartenant à A_{i+1} pour être exécuté sur les architectures instanciées à partir de ce niveau. Ceci est comparable à la définition des classes et des objets dans le paradigme objet, où les structures et les comportements des objets sont toujours décrits par leurs classes mais s'appliquent aux objets.

On peut dire que SAEM offre une mécanique opératoire avancée. L'architecte a à sa disposition les relations d'instanciation, de spécialisation, de composition et d'utilisation pour pouvoir décrire et gérer l'évolution selon ses besoins. En outre, en associant à chaque type d'élément architectural son ou ses styles d'évolution, le raisonnement relatif à l'évolution se retrouve distribué à travers les éléments de l'architecture. En règle générale, un raisonnement distribué facilite le passage à l'échelle, qui est une question importante en raison de la taille et la complexité croissantes des systèmes logiciels, et qui est prônée dans les définitions récentes de l'évolution comme celles trouvées dans les architectures auto-organisées [GMK02]. Nous verrons dans le Chapitre 4 que le raisonnement supporté par SAEM est de type classificatoire.

Niveau 3 : SAEM s'efforce d'offrir un bon niveau de réutilisation. En effet, il favorise :

1. L'extensibilité : chaque nouvel ajout de style à un élément évolutif ne perturbe aucunement le fonctionnement des styles déjà présents.
2. L'évolutivité : la compétence d'un style peut être modifiée sans impacter la façon dont d'autres styles l'utilisent.
3. La compositionalité : la relation de composition fournie par SAEM permet la spécification de styles d'évolution composites.
4. La remplaçabilité : un style peut tout à fait être substitué à un autre style, du moment qu'il possède la même entête.

SAEM offre un bon support pour le développement d'architectures évolutives, en reposant sur une description minimaliste des styles d'évolution. Le formalisme du modèle en Y contribue à la simplicité de SAEM. Enfin, l'adaptabilité n'est pas prise en compte pour l'instant par le modèle d'évolution SAEM, mais a été anticipée à travers la notion de vue générique (cf. figure 3.13).

3.7 Conclusion

Nous avons présenté dans ce chapitre le modèle d'évolution baptisé SAEM, que nous proposons pour prendre en compte la problématique de l'évolution structurelle des architectures logicielles. Nous avons mis en avant dans SAEM la nécessité de dissocier la spécification de l'évolution de la spécification de l'architecture logicielle en elle-même. C'est ainsi que SAEM s'est efforcé de proposer des concepts pour la spécification et la gestion de l'évolution d'une architecture logicielle, indépendamment de tout ADL. Nous avons souligné au travers de SAEM, la nécessité de considérer les différents niveaux de modélisation d'une architecture logicielle, et la nécessité de gérer l'évolution au travers de ces différents niveaux, de manière uniforme. Nous avons également mis en avant l'importance de la propagation des impacts pour garantir la cohérence globale de l'architecture amenée à évoluer. Les stratégies de propagation sont véhiculées par les relations sémantiques qui existent entre les styles d'évolution.

Nous avons introduit le formalisme MY qui décrit les concepts du styles d'évolution selon trois branches : domaine, entête, compétence. Des liens intra et inter permettent de gérer les

différentes dépendances qu'entretiennent les trois aspects du tryptique. En outre, MY est multi-vues et multi-abstractions. Ainsi, MY est utilisé pour décrire sous forme modulaire et réutilisable des styles d'évolution.

Jusqu'à présent, nous n'avons proposé aucune démarche pour aider les architectes à exploiter SAEM. Une démarche est pourtant nécessaire pour assurer l'efficacité de notre approche de réutilisation dans le contexte de l'évolution dans les architectures à base de composants. Cet aspect fait l'objet du chapitre suivant.

Chapitre 4

Bibliothèques pour les styles d'évolution

Contrairement aux approches d'évolution usuelles dont la réalisation démarre à partir de rien – « from scratch » – et nécessite de tout réinventer à chaque fois, la réutilisation est une approche qui peut permettre de traiter l'évolution d'une architecture logicielle à partir d'éléments existants stockés dans des bibliothèques. De cette façon, les architectes minimisent le travail redondant. Ils augmentent la fiabilité de leur travail vu que chaque évolution réutilisée a déjà été contrôlée et inspectée au cours de sa spécification originelle. Ainsi, le temps consacré aux activités d'évolution dans une architecture est réduit de façon considérable. L'idée développée ici est celle de véritables styles d'évolution sur étagère. Ainsi, nous suggérons de désigner ces styles prêt à l'emploi sous l'acronyme EOTS (*Evolution-Off-The-Shelf*).

Il n'est donc pas suffisant de disposer d'un modèle d'évolution riche et expressif, encore faut-il lui associer les bons mécanismes et les bonnes règles pour pouvoir réutiliser les styles d'évolution. Dans ce chapitre nous donnons les instruments nécessaires à la mise en œuvre d'une approche par réutilisation dans le contexte de SAEM. En particulier, le modèle en Y présenté dans le chapitre précédent est désormais doté de trois niveaux de modélisation et améliore la réutilisation des styles d'évolution en supportant des bibliothèques pour chaque niveau. Nous identifions différents acteurs pour intervenir dans les bibliothèques et nous développons un processus de construction de ces dernières « pour la réutilisation » et « par la réutilisation ». Nous mettons l'accent sur la bibliothèque stockant des types de styles d'évolution et offrons une infrastructure pour les exploiter, reposant sur une technique de raisonnement de nature classificatoire. Ce type de raisonnement se base sur les processus de classification et s'appréhende comme une procédure de déduction opérant sur une hiérarchie quelconque (de spécialisation, de composition, de modélisation). Notre infrastructure inclut deux types de classification, l'une opérant au niveau statique, l'autre au niveau dynamique. La première concerne la classification de types de styles d'évolution et repose sur l'inférence de relations "est-une-sorte-de" ou "est-composé-de". Elle sera exploitée pour peupler et interroger la bibliothèque de types de styles. La seconde concerne la classification d'instances de styles d'évolution et repose sur l'inférence de relations "est-un". Elle sera exploitée dans le cadre de l'instanciation de types de styles afin de déterminer, si cela est nécessaire, les bonnes compétences à appliquer sur une architecture en cours d'évolution.

4.1 Bibliothèques pour les styles d'évolution

Nous avons doté MY de trois niveaux de modélisation, supportant des bibliothèques d'éléments réutilisables pour chaque niveau. Afin de clarifier le fonctionnement du processus de réuti-

lisation et l'intervention de différents acteurs sur les bibliothèques, nous décrivons la construction des bibliothèques.

4.1.1 Niveaux de modélisation des bibliothèques

MY supporte des bibliothèques d'éléments de différents niveaux de modélisation, à partir de la définition du méta-modèle SAEM jusqu'au niveau de l'évolution à l'exécution (cf. figure 4.1). Nous pouvons distinguer trois niveaux : le niveau méta-style d'évolution E2, le niveau type de style d'évolution E1 et le niveau instance de style d'évolution E0. Ainsi, la bibliothèque de niveau méta stocke des éléments méta tels que le domaine, l'entête et la compétence ; la bibliothèque de niveau type stocke des types d'éléments tels que *C&C*, *IMovePort* et *MovePortImpl* ; et enfin la bibliothèque de niveau instance stocke des éléments tels que l'évolution visant spécifiquement à déplacer le port `p1`.

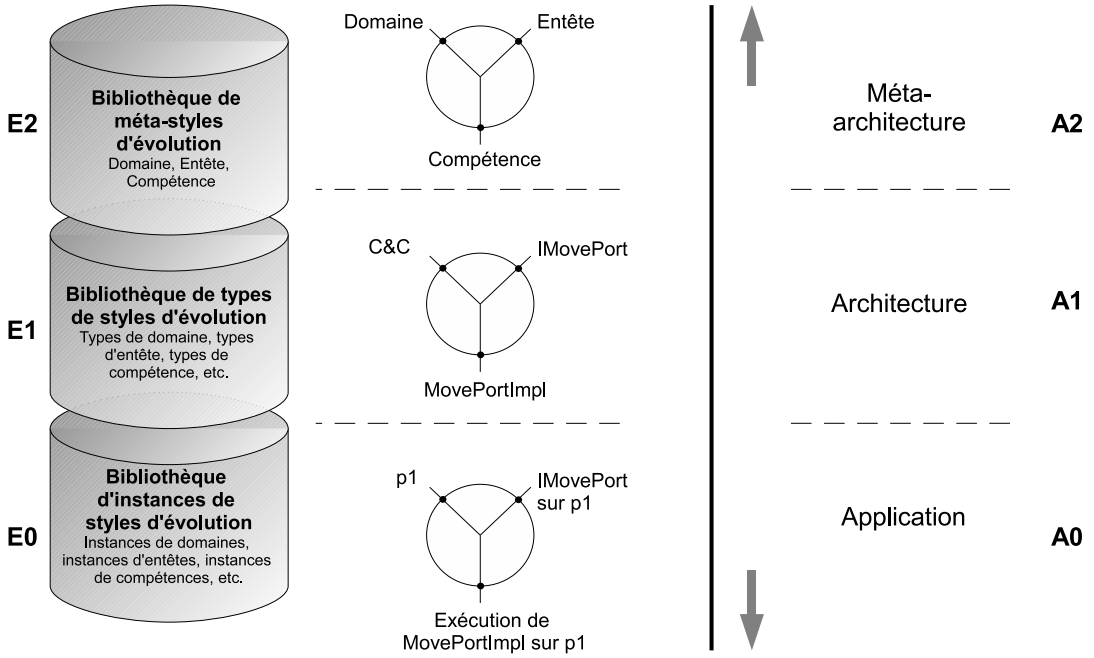

Figure 4.1 – Les bibliothèques pour les styles d'évolution.

Notre approche permet d'identifier les éléments réutilisables pour chaque niveau de modélisation, de manière uniforme en utilisant un MY. Ces éléments sont alors stockés dans une bibliothèque liée à chaque niveau de modélisation et utilisée par son niveau inférieur. Par exemple, les éléments réutilisables du niveau méta-style sont : domaine, entête et compétence. Ils sont stockés dans une bibliothèque de niveau E2 et utilisés au niveau E1.

Par ailleurs, la Figure 4.1 tend à l'illustrer la coexistence des trois niveaux de modélisation du paradigme des styles d'évolution avec les trois (ou quatre) niveaux de modélisation des architectures logicielles qui peuvent "glisser" verticalement afin de préciser sur quel niveau de modélisation une évolution est opérée. Par exemple, une bibliothèque de niveau E1 peut contenir des types de styles d'évolution qui peuvent être spécifiés sur la base des éléments du domaine du niveau A2 ou bien A1 d'une architecture logicielle. Selon le cas, le niveau A1 ou le niveau A0

d'une architecture peut évoluer concrètement grâce aux instances des styles d'évolution contenues au niveau E0.

4.1.2 Acteurs relatifs aux bibliothèques

Nous avons identifié trois rôles d'acteurs pour le modèle en Y. Chacun est associé à un niveau donné (E2, E1 ou E0). En plus, un quatrième acteur est nécessaire pour gérer les bibliothèques. Ainsi, nous avons :

- Le constructeur d'infrastructure d'évolution (CIE) : définit les concepts de base sur lesquels l'architecte d'application évolutive s'appuie pour établir un ensemble de types de styles d'évolution. Le CIE est concerné par le niveau de modélisation E2.
- L'architecte d'application évolutive (AAE) : son rôle est de concevoir et de décrire les types de styles d'évolution qui seront utilisés pour le développement d'applications évolutives. Il le fait à travers l'instanciation des concepts fournis par le CIE. L'AAE est concerné par le niveau de modélisation E1.
- Le développeur d'application évolutive (DAE) : son rôle est de faire évoluer une architecture logicielle en instanciant le niveau E1. Le DAE est concerné par le niveau E0.
- Le bibliothécaire de l'évolution (BE) : construit et gère toute les bibliothèques. Le bibliothécaire de l'évolution est concerné par tous les niveaux de modélisation. Les bibliothèques sont donc hiérarchisées suivant les trois niveaux de modélisation et se composent :
  - au niveau E2 des concepts de domaine, d'entête et de compétence définis par le CIE,
  - au niveau E1 des types de styles d'évolution, créés par assemblage de types de domaines, de types d'entêtes et de types de compétences définis par l'AAE,
  - au niveau E0 des instances de styles d'évolution définies par le DAE, créées à partir d'instances de domaines, d'instances d'entêtes et d'instances de compétences.

La Figure 4.2 présente le diagramme de séquence d'un scénario possible pour les différentes opérations sur les bibliothèques. Ce scénario se décompose en deux étapes :

1. Étape de définition d'un type de style d'évolution : au cours de cette étape, l'AAE envoie une demande des éléments du niveau E2 au BE pour pouvoir définir un type de style (phase 1). Ensuite, le BE recherche les éléments dans la bibliothèque du niveau E2, les récupère (phases 2 et 3) et les éléments sont fournis à l'AAE (phase 4). Enfin, l'AAE confirme la réception par un accusé (phase 5). Dans le cas où ces éléments ne sont pas disponibles dans la bibliothèque, le BE demande au CIE de construire ces éléments.
2. Étape d'évolution d'une architecture logicielle : durant cette étape, le DAE demande au BE des types de styles pour faire évoluer l'architecture (phase 6). Ensuite le BE recherche dans la bibliothèque de niveau E1, récupère ces types de styles (phases 7 et 8) et fournit ces éléments au DAE (phase 9). Enfin, le DAE envoie au BE un accusé de réception (phase 10). Dans le cas où ces éléments ne sont pas disponibles dans la bibliothèque, le BE demande à l'AAE de les construire.

4.1.3 Construction des bibliothèques

Nous l'avons vu, différentes bibliothèque d'éléments réutilisables existent pour chaque niveau de modélisation. La problématique de réutilisation se pose alors en terme de construction de ces

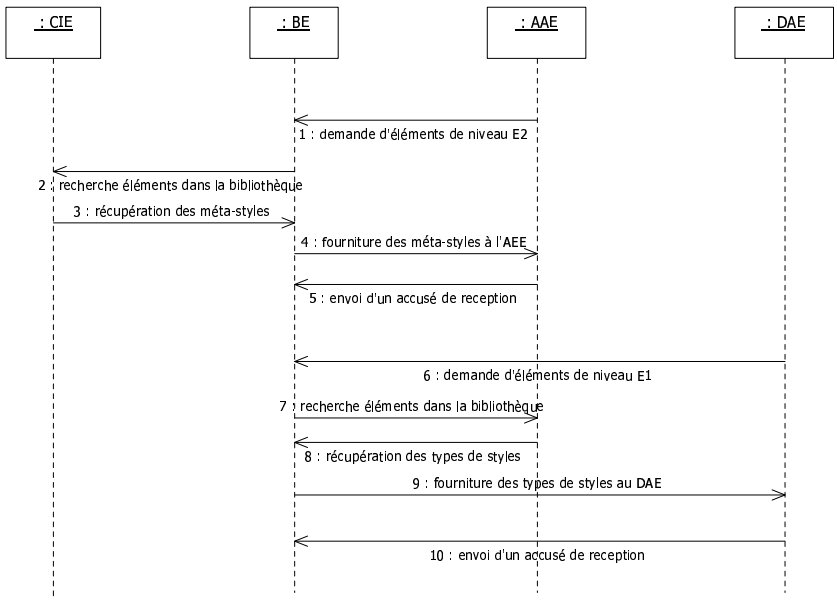

Figure 4.2 – Le diagramme de séquence pour les opérations sur les bibliothèques.

bibliothèques. Nous suggérons une approche de construction pour et par la réutilisation pour chaque niveau. Une telle approche appliquée spécifiquement aux bibliothèques de types de styles d'évolution (niveau E1) fait l'objet d'une attention toute particulière pour notre travail.

4.1.3.1 Processus pour et par la réutilisation

Nous suggérons une approche permettant aux éléments réutilisables de chaque niveau (E2, E1 et E0) d'être vus selon leur construction (pour la réutilisation) afin de construire une bibliothèque, et leur utilisation en les choisissant à partir d'une bibliothèque (par la réutilisation) pour établir de nouveaux éléments ou pour les instancier. Cette approche concerne la description des éléments réutilisables suivants deux étapes [CS99, BCO*04] :

1. L'ingénierie pour la réutilisation qui permet de spécifier les éléments qui sont établis pour la réutilisation de chaque niveau de modélisation. Les éléments réutilisables sont identifiés selon leur niveau de modélisation, représentés en utilisant un langage formel, semi-formel ou descriptif et ensuite intégrés dans une bibliothèque associée en choisissant une méthode bien définie (hiérarchisée par type, par taille, par contenu, par la fréquence de réutilisation, etc.).

2. L'ingénierie par la réutilisation qui permet de spécifier les éléments qui peuvent être établis par la réutilisation où la recherche et la sélection des éléments réutilisables sont faites selon les besoins, qui sont exprimés par les différentes catégories d'acteurs. La technique de recherche dépend fortement de l'organisation des éléments dans la bibliothèque aussi bien que du langage choisi durant l'étape précédente. Les éléments choisis sont adaptés si besoin, et utilisés dans le niveau de modélisation en cours.

4.1.3.2 Activités spécifiques à la réutilisation de styles d'évolution

Dans cette section, sont considérées les activités spécifiques à la mise en œuvre d'une approche de réutilisation de styles d'évolution, stockés dans une bibliothèque de niveau E1. Ces activités se rangent dans deux catégories, celles correspondant au point de vue du producteur de style d'évolution (processus pour la réutilisation) et celles correspondant au point de vue du consommateur de style d'évolution (processus par la réutilisation).

Production de style d'évolution Cette activité est essentielle dans le processus de réutilisation puisqu'elle a pour objectif de produire les styles d'évolution réutilisables. C'est aux architectes d'application évolutive (AAE) que revient la tâche d'identifier de manière systématique toutes les évolutions qui peuvent être candidates à la réutilisation, à travers leur connaissance du domaine et leurs expériences passées. Après leur identification, elles doivent être représentée dans notre langage de style. Les styles sont alors organisés selon des liens intra-concepts (spécialisation, composition et utilisation) pour être intégrés dans une bibliothèque de niveau E1.

Consommation de style d'évolution Cette activité consiste à rechercher dans une bibliothèque un style d'évolution qui réponde à un besoin particulier d'évolution. Dans la plupart des situation, il n'y a pas un seul style candidat qui répond au besoin, mais plusieurs. Les techniques de recherche mise en oeuvre dans ce contexte peuvent être issues du domaine de la recherche d'information (navigation dans une hiérarchie, recherche par mot-clés). Nous distinguons deux façons de réutiliser un style d'évolution :

- La réutilisation à travers la composition et la spécialisation : un AAE récupère un ou plusieurs styles existants dans une bibliothèque afin de les adapter à ses besoins par composition et/ou par spécialisation. Notamment, les styles abstraits sont des super-styles dans lesquels la compétence n'a pas été implémentée. Ainsi, un style abstrait définit un entête standard pour tous ses styles descendants. Il forme une sorte de point d'extension destiné à être complété lors de sa réutilisation.
- La réutilisation à travers l'instanciation : la réutilisation devient effective si le style peut être instancié et exécuté par un DAE sur son architecture en cours d'évolution. Ici, la réutilisation se traduit par un processus de génération [BR89]. Contrairement aux connaissances de type produit, il y a une forte différence entre le style d'évolution avant sa réutilisation et après sa réutilisation (le résultat de l'évolution sur l'architecture).

4.1.3.3 Approche orienté problème du style d'évolution

Un style d'évolution respecte pleinement le principe d'abstraction, jugé essentiel en matière de réutilisation [CS99]. En effet, un style d'évolution reconnaît la distinction entre la spécification et la réalisation de l'évolution, à travers son entête et sa compétence. Comme l'illustre la Figure 4.3 on peut considérer que l'entête d'un style est centré sur l'expression d'un problème et que la compétence correspond à sa solution. Clairement, les styles et les patrons sont deux types de propositions représentatives de l'approche orienté problème.

Nous considérons toutefois le concept de style comme une variante plus formelle du patron, notamment à travers son mécanisme d'instanciation, plus rigoureux que l'opération "intuitive" d'imitation portant sur les patrons. En outre, la faible quantité d'informations informelles dé-

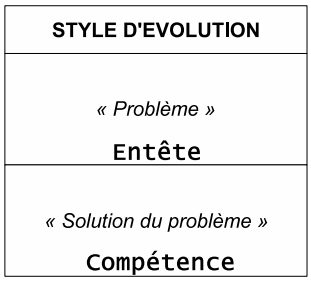

Figure 4.3 – Orientation problème d'un style d'évolution.

crites au sein d'un style permet plus qu'une simple utilisation manuelle. Les styles d'évolution structurent les solutions sur la base d'un triplet domaine-entête-compétence et, à l'instar des patrons, ils peuvent être vus comme des *codes de bonnes pratiques*. Une bibliothèque de styles d'évolution doit ainsi être perçue comme un réservoir de solutions à des problèmes d'évolution récurrents sur les architectures logicielles, alimenté par des architectes d'applications évolutives et où viennent puiser d'autres architectes ou développeurs d'applications évolutives.

4.2 Élaboration d'une bibliothèque de styles d'évolution

L'élaboration d'une bibliothèque de styles d'évolution cible des problèmes d'organisation hiérarchique des styles d'évolution d'une part, et de définition des besoins de recherche, d'accès, d'adaptation et d'instanciation des styles d'évolution, d'autre part. Elle produit une infrastructure pour le système de réutilisation, en rendant opérationnelles les opérations d'intégration et de recherche de styles d'évolution dans la bibliothèque de niveau E1. Un même raisonnement classificatoire sur les types de styles sous-tend ces opérations. C'est pourquoi nous commençons par expliquer son fonctionnement avant de présenter les opérations en elle-mêmes.

4.2.1 Classification de styles d'évolution

Tout au long de son existence, l'être humain apprend une grande quantité de connaissance qu'il mémorise de façon structurée. Face à une situation inconnue, il cherche dans sa mémoire des situations similaires déjà vécues pour en tirer des conclusions. Ce raisonnement entraîne donc une étape de classification dans laquelle la nouvelle connaissance est comparée avec les apprentissages précédents et classée dans la structure. Dans le domaine de l'intelligence artificielle, Clancey [Cla85] a montré que la plupart des systèmes expert incluent des étapes de classification. La définition du terme classification qui nous intéresse ici est la classification de définitions de styles d'évolution dans la bibliothèque de niveau E1, jouant ici le rôle de base de connaissance. Pour mettre en pratique ce processus, nous nous appuyons sur l'organisation des styles d'évolution stockés sous forme d'un graphe.

4.2.1.1 Un double point de vue : spécialisation et composition

Nous définissons le contenu de la bibliothèque de niveau E1 comme un graphe quelconque de styles d'évolution, qui peut être filtré à travers le point de vue de la spécialisation et de la composition. Ceci est possible car nous considérons explicitement la nature hiérarchique des relations sémantiques de spécialisation et de composition (cf. section 3.3.4). Le filtrage à travers ces deux points de vues isole deux hiérarchies distinctes l'une de l'autre. Nous schématisons cela par la Figure 4.4. Examinons maintenant plus en détail ces deux types de hiérarchie.

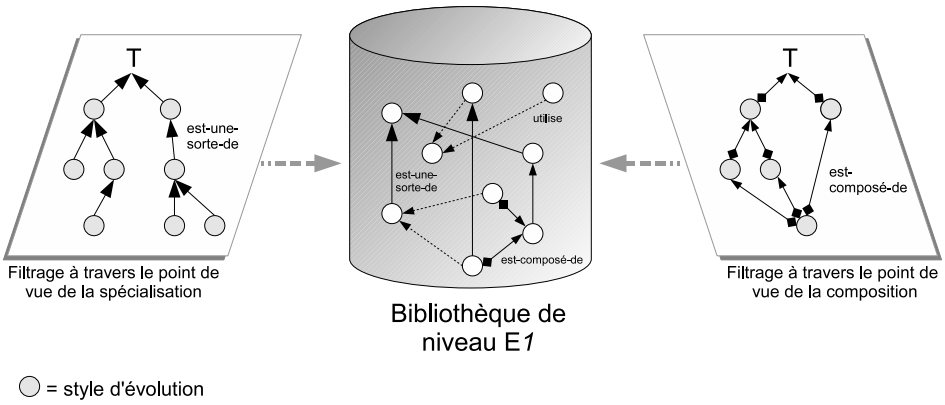

Figure 4.4 – Filtrage par points de vue sur la bibliothèque de niveau E1.

Hiérarchie de spécialisation Le filtrage par la relation de spécialisation produit un arbre[1], appelé hiérarchie de spécialisation. Nous notons cette hiérarchie $H_s = (E, \top_s, \sqsubseteq_s)$, où E est un ensemble fini de styles d'évolution, $\sqsubseteq_s$ est la relation d'ordre de spécialisation entre les styles, et $\top_s$ est l'élément d'unité de E suivant $\sqsubseteq_s$. $\top_s$ est appelée racine de la hiérarchie de spécialisation et correspond à un style d'évolution abstrait et d'entête vide, introduit artificiellement par le système, nommé *Evolution*. On représente le fait qu'un style d'évolution B est subsumé par A en traçant l'arc $\overrightarrow{BA}$. La présence de cet arc signifie que B est une sorte de A.

Hiérarchie de composition Le filtrage par la relation de spécialisation produit un graphe, appelé hiérarchie de composition. Nous notons cette hiérarchie $H_c = (E, \top_c, \sqsubseteq_c)$, où E est un ensemble fini de styles d'évolution, $\sqsubseteq_c$ est la relation d'ordre de composition entre les styles, et $\top_c$ est l'élément d'unité de E suivant $\sqsubseteq_c$. $\top_c$ est appelée racine de la hiérarchie de composition et correspond à un style d'évolution introduit artificiellement par le système, nommé *NEV* (No EVolution), dont l'entête et la compétence sont vides. On représente le fait qu'un style d'évolution B est subsumé par A en traçant l'arc $\overrightarrow{BA}$. La présence de cet arc signifie que B est composé de A.

4.2.1.2 Hiérarchie de subsomption

Une relation de subsomption est une relation générale qui définit une structure hiérarchique parmi un ensemble de concepts. Elle ne définit pas forcément une arborescence, mais plutôt un

1. Car SAEM n'autorise que l'héritage simple.

treillis. Un subsumant et un subsumé se disent respectivement d'un concept hiérarchiquement supérieur et d'un concept hiérarchiquement inférieur, selon une relation sémantique particulière. Par exemple, on peut établir qu'un entier subsume un autre entier s'il est un multiple de ce dernier. Un test de subsomption est une opération qui infère l'existence d'un relation de subsomption entre deux concepts B et A (ce qui est noté B $\sqsubseteq$ A). Un test de subsomption est donc une fonction booléenne de signature $B \times A \rightarrow Bool$.

Le processus de classification de styles cherche à mettre en évidence les liens implicites qui existent entre les styles d'évolution. Ce mode de raisonnement, appelé raisonnement classificatoire ou taxinomique, permet d'exploiter la connaissance capitalisée dans une bibliothèque, relativement à la hiérarchie étudiée. On peut définir les relation de spécialisation et de composition de SAEM comme deux relations de subsomption spécifiques. Notre stratégie est de proposer un processus capable de reproduire un raisonnement par classification, et totalement indépendant de la hiérarchie considérée. La caractère générique de ce processus permet une économie lors de la phase de codage, dont le comportement pourra être paramétré au besoin par :

- Le test de subsomption par la spécialisation (calcule la relation "est-une-sorte-de")
- Le test de subsomption par la composition (calcule la relation "est-composé-de")

4.2.1.3 Raisonnement classificatoire

Indépendamment du test de subsomption considéré, le raisonnement par classification mené sur un style d'évolution x se décompose en deux étapes, dont le lecteur trouvera le détail en Annexe B :

1. La recherche des subsumants les plus spécifiques (*SPS*) du style d'évolution x à classer. Cela revient à déterminer l'ensemble de ses ascendants immédiats dans la hiérarchie.
2. La recherche des subsumés les plus généraux (*SPG*) du style d'évolution x à classer. Cela revient à déterminer l'ensemble de ses descendants immédiats dans la hiérarchie.

Un exemple du résultat du calcul des *SPS* et des *SPG* d'un style x est illustré par la Figure 4.5, soit sur une hiérarchie de spécialisation, soit sur une hiérarchie de composition. Lorsque l'on considère une description de style d'évolution x à classer dans une bibliothèque, celle-ci possède nécessairement au moins deux relation sémantiques avec l'existant. En effet, par défaut, toute nouvelle description est reliée par spécialisation à la racine *Evolution*[2], et reliée par composition à la racine *NEV*. Bien sûr, la nouvelle description peut parfaitement être reliée à des styles situés autre part dans les hiérarchies de spécialisation et de composition.

4.2.2 Opération d'intégration dans une bibliothèque

L'architecte d'application évolutive (AAE) a la charge de spécifier les éléments qui sont établis pour la réutilisation. Ces éléments réutilisables sont identifiés et représentés sous forme de styles et ensuite intégrés dans la bibliothèque du niveau E1, dans le respect de sa structure hiérarchique. Pour rendre effective cette intégration, nous décrivons un processus de peuplement de la bibliothèque basé sur un raisonnement classificatoire.

2. Comme dans le langage Java par exemple, où toute nouvelle classe étend par défaut la classe racine `Object`.

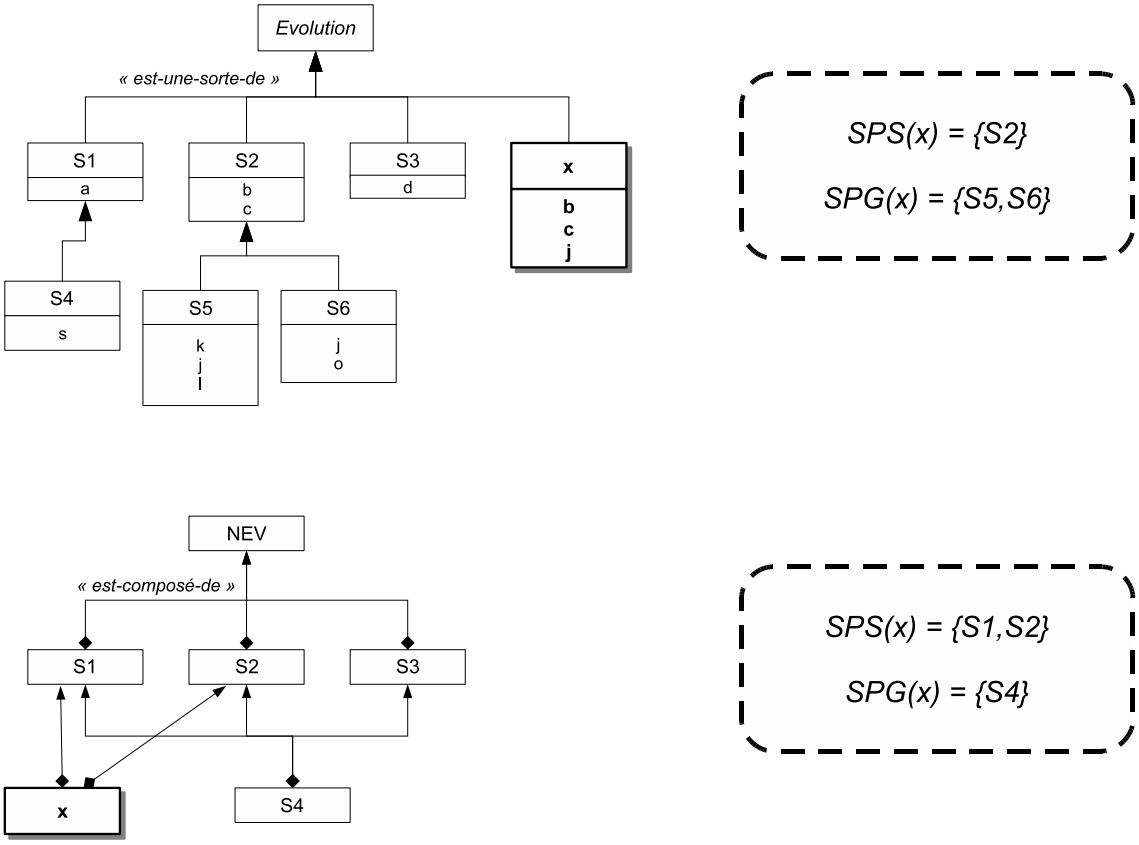

Figure 4.5 – Représentation schématique du raisonnement classificatoire sur un style d'évolution x dans une hiérarchie de spécialisation (en haut) et dans une hiérarchie de composition (en bas).

4.2.2.1 Objectif

Tout l'enjeu de l'intégration consiste à préserver la structuration de la bibliothèque de niveau E1 selon ses deux points de vues : spécialisation et composition. Puisque le contenu de la bibliothèque est hiérarchisé, c'est une condition sur toute la bibliothèque. En d'autres termes, chaque fois que l'on fait quelque chose, comme ajouter un style d'évolution, il faut réorganiser la bibliothèque pour maintenir la condition globale [3] induite par les relations d'ordre de spécialisation et de composition. La suppression d'un style d'évolution de la bibliothèque est soumise aux mêmes règles mais n'est pas abordée dans le cadre de cette thèse.

L'opération d'intégration est d'autant plus importante dans un contexte multi-utilisateurs, où différents architectes d'applications évolutives peuvent coopérer pour augmenter le niveau global d'expertise. En effet, un AAE n'a souvent qu'une vision locale des connaissances contenues dans la bibliothèque. Par conséquent, disposer d'une infrastructure pour la maintenance des relations sémantiques entre les styles d'évolution est important. A ce titre, il est utile de rappeler que ces relations préservent la cohérence des connaissances accumulées, et qu'elles sont remises en cause chaque fois que l'intégration d'un nouveau style d'évolution dans la bibliothèque se produit. Un processus uniquement assuré par l'humain peut être très dangereux, voire impossible, dès lors que la taille de la bibliothèque atteint un certain seuil. Nous pouvons schématiser le fonctionnement du processus d'intégration d'un style d'évolution x dans une bibliothèque E de la manière suivante :

$$\boxed{(E, \top, \sqsubseteq) \times \{x\} \rightarrow (E \cup x, \top, \sqsubseteq)}$$

4.2.2.2 Fonctionnement

Bien que les problématiques de classification soient différentes entre spécialisation et composition, le schéma général d'intégration d'un style x dans la bibliothèque est le suivant :

1. Calcul de SPS(x), l'ensemble des styles subsumants x dans la bibliothèque.
2. Calcul de SPG(x), l'ensemble des styles subsumés par x dans la bibliothèque.
3. Ajout des relations entre x et ses *SPS* et *SPG*.
4. Suppression des relations de transitivité éventuellement créées.
5. Vérification de la conservation des propriétés de la hiérarchie et modification éventuelle.

Les étapes 1 et 2 ont été expliquées précédemment. Les étapes 3 à 5 concernent l'insertion proprement dite de la spécification de x dans la hiérarchie considérée. Le processus d'insertion peut être amené à restructurer la spécification du style x et celle de ses voisins. Pour s'assurer d'une intégration en une seule passe, le processus d'insertion dans la hiérarchie de composition doit être antérieur au processus d'insertion dans la hiérarchie de spécialisation. De cette façon, la variation des relations de type "est-composé-de" induite par une restructuration de la spécification x est prise en compte quand vient le moment de classer x dans la hiérarchie de spécialisation. Relié à ce dernier point, l'étape 5 concerne spécifiquement la hiérarchie de spécialisation où il est question d'éliminer automatiquement la définition de certains paramètres et assertions de l'entête des spécifications restructurées pour éviter la redondance avec celles désormais héritées.

3. Le maintien d'une condition globale est un exemple d'un calcul orienté but.

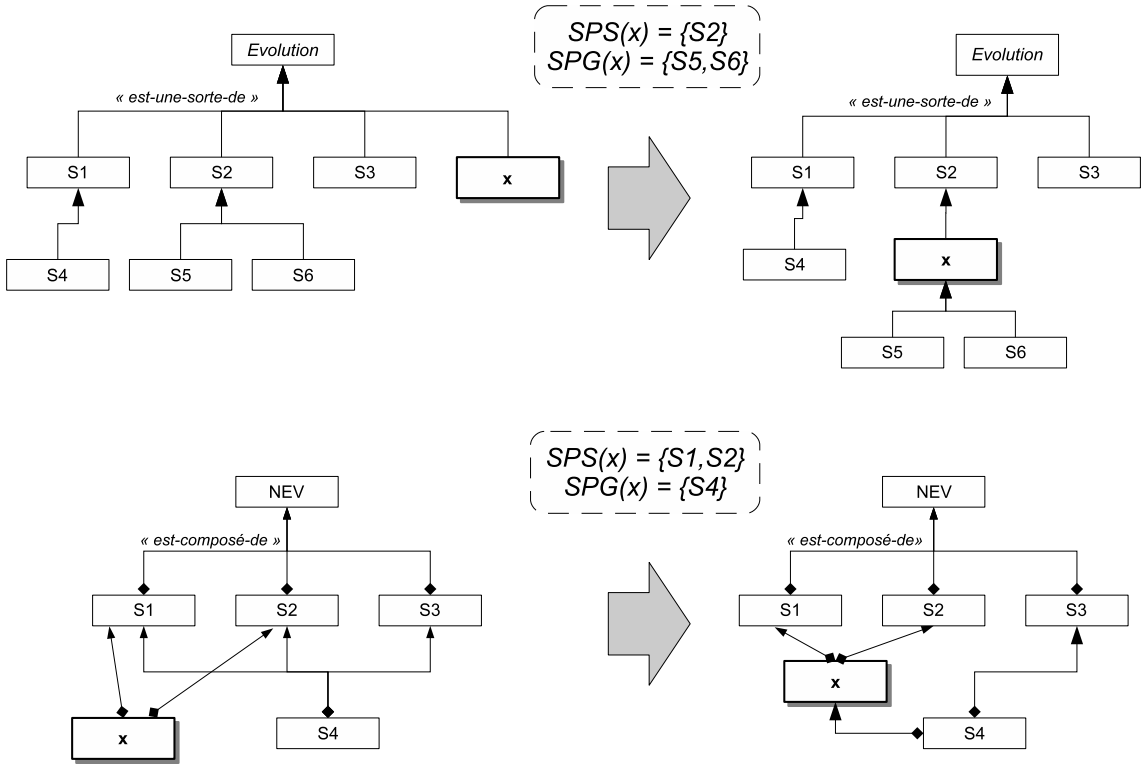

Figure 4.6 – Schéma de l'opération d'intégration d'un style x dans une hiérarchie de spécialisation (en haut) et dans une hiérarchie de composition (en bas).

En reconsidérant l'illustration présentée dans la section 4.2.1.3, le résultat de l'insertion du style x serait celui de la Figure 4.6. Selon le point de vue de la spécialisation, et compte tenu des *SPS* et *SPG* calculés par un raisonnement classificatoire basé sur la subsomption de spécialisation, x est une sorte de *S2*, et *S5* et *S6* sont des sortes de x. Selon le point de vue de la composition, et compte tenu des *SPS* et *SPG* calculés par un raisonnement classificatoire basé sur la subsomption de composition, x est composé de *S1* et de *S2* (*i.e.*, pas de restructuration nécessaire) et *S4* est maintenant composé de x et de *S3*. Ces restructurations en cascades n'altèrent pas les buts des styles stockés dans la bibliothèque mais leur assurent une bonne organisation qui influencera nécessairement leur recherche.

4.2.3 Opération de recherche dans une bibliothèque

L'architecte d'application évolutive (AAE) ou le développeur d'application évolutive (DAE) ont la charge de spécifier les éléments qui peuvent être établis par la réutilisation dans la bibliothèque de niveau E1. Les styles d'évolution doivent être recherchés et sélectionnés selon les besoins. Pour opérationnaliser cette recherche, l'infrastructure offre de nouveau une technique basée sur un raisonnement classificatoire.

4.2.3.1 Objectif

L'objectif est d'explorer la bibliothèque en vue de réutiliser, autant que possible, les styles d'évolution qui répondent aux besoins exprimés par un AAE/DAE : sa question (ou requête). Sa question correspond à un problème d'évolution auquel il espère trouver une réponse par le biais de l'infrastructure de réutilisation. Répondre à sa question consiste à trouver dans la bibliothèque le ou les style(s) d'évolution qui offre(nt) la compétence demandée. Une technique de recherche par mot-clés appliquée au nom et/ou au but des styles n'est pas pertinente ici. A la place, nous suggérons d'exploiter un raisonnement classificatoire pour déterminer la place d'une requête dans la structure hiérarchique de la bibliothèque. Toutefois, cette approche nécessite de se donner la peine de représenter une requête comme un style d'évolution q. En contrepartie, il n'est pas nécessaire pour l'utilisateur d'apprendre un langage spécifique de requête (type SQL ou autre).

La requête q est représentée sous forme d'un style abstrait, c'est-à-dire qui possède un entête, mais pas de compétence. En effet, l'architecte est capable de décrire un problème mais n'est pas en mesure de décrire sa solution. Or, c'est précisément cette solution que l'architecte peut raisonnablement espérer obtenir d'une infrastructure de réutilisation. Comme l'illustre la Figure 4.7, la phase de résolution du problème peut en général être appréhendée comme la complémentation d'un entête par une compétence en vue d'obtenir un style d'évolution concret (car complet).

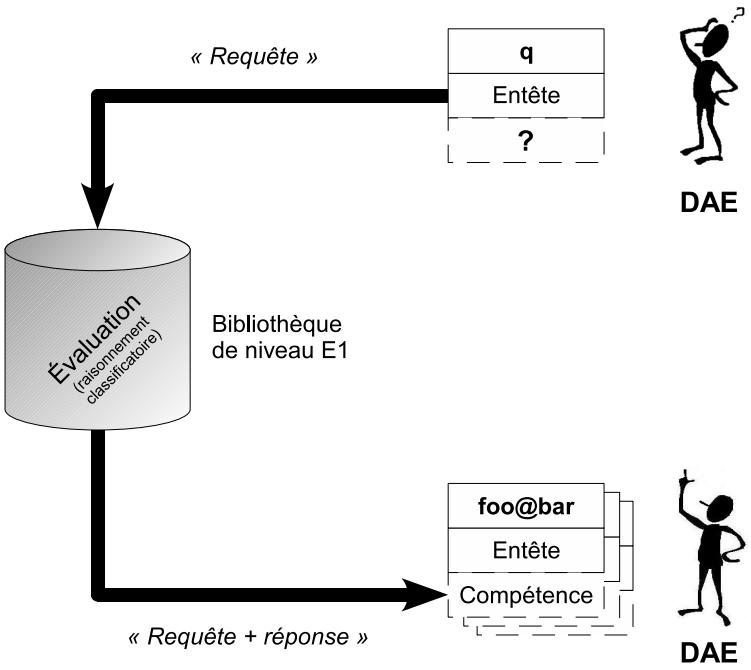

Figure 4.7 – Vue schématique du fonctionnement de l'opération de recherche dans la bibliothèque à partir d'un style abstrait q.

Un raisonnement classificatoire peut être employé comme mécanisme d'évaluation de la requête puisque la partie entête d'un style est nécessaire et suffisante pour le test de subsomption. Le processus d'évaluation d'une requête consiste ainsi à situer q dans une bibliothèque E et permet d'aller au delà de simples réponses binaires - "oui" ou "non" -, notamment en prenant en considération des réponses candidates. Les résultats de la requête, notés R, représentent l'en-

semble des compétences stockées dans la bibliothèque et en mesure de satisfaire la requête q, directement ou à travers une adaptation. Nous pouvons schématiser le fonctionnement du processus de recherche de la manière suivante :

$$\boxed{(E, \top, \sqsubseteq) \times \{q\} \rightarrow R}$$

4.2.3.2 Fonctionnement

Le processus d'évaluation d'une requête renvoi trois valeurs possibles : succès, candidat, ou échec. Nous détaillons les différentes situations dans la table 4.1.

Évaluation	Raisonnement	Description
Succès	*Spécialisation*	un résultat satisfait exactement la requête q. Ce cas intervient lorsque SPS(q) = SPG(q) selon le point de vue spécialisation.
	Composition	un résultat satisfait exactement la requête q. Ce cas intervient lorsque SPS(q) = SPG(q) selon le point de vue composition.
Candidat	*Spécialisation*	les SPS(q) calculés selon le point de vue de la spécialisation représentent des évolutions plus générales qui peuvent se substituer à la requête q.
	Composition	les SPS(q) calculés selon le point de vue de la composition représentent des évolutions ayant une plus fine granularité que la requête q, et sont destinés à être réutilisés "ensembles".
Échec	*Spécialisation*	aucun résultat ne satisfait la requête q. Ce cas intervient lorsque les SPS(q) et SPG(q) sont vides selon le point de vue spécialisation.
	Composition	aucun résultat ne satisfait la requête q. Ce cas intervient lorsque les SPS(q) et SPG(q) sont vides selon le point de vue composition.

Table 4.1 – Classes de résultats basés sur un raisonnement classificatoire par spécialisation ou composition.

La technique s'appuyant sur un raisonnement classificatoire supporte le couple succès/échec offert par n'importe quelle technique classique d'entreposage, mais avec tout de même la possibilité de différencier son raisonnement : spécialisation ou composition. De ce fait, l'infrastructure de réutilisation répond de deux manières différentes : « votre problème d'évolution est une sorte de ... » ou « votre problème d'évolution est composé de ... ». Le succès signifie l'équivalence [4] de la requête q avec un style présent dans la bibliothèque. Remarquons que l'équivalence selon la spécialisation implique l'équivalence selon la composition. L'échec signifie que la bibliothèque ne contient pas encore de solution au problème posé. Assurément, la valeur ajoutée de la technique à base de classification est de proposer des réponses candidates qui auraient été normalement ignorées par une technique d'entreposage classique. Disposer de solutions alternatives est un avantage car la probabilité d'appariement strict diminue dans un contexte multi-utilisateur comme c'est le cas ici.

Enfin, nous faisons remarquer que l'uniformité de l'approche proposée permet un basculement rapide de l'opération de recherche vers l'opération d'intégration. Ceci présente un intérêt dans deux cas précis :

4. Deux styles sont équivalents si une relation de subsomption mutuelle existe entre eux, *i.e.* $A \equiv BsiA \sqsubseteq B \wedge B \sqsubseteq A$

- Le premier cas de figure se produit lorsque des styles candidats sont destinés à être réutilisés "ensembles" par l'architecte. L'instanciation de ces styles sur les bons éléments architecturaux et dans le bon ordre témoigne de la compétence de l'architecte. On en revient alors à la motivation première de cette thèse, stipulant qu'il serait préférable de capturer cette compétence de manière explicite pour en faire un nouveau style composite, réutilisable à son tour. Le cas échéant, le DAE se retrouve dans la position de l'AAE.
- Le second cas de figure apparaît lorsque les styles souhaités par le DAE ne sont pas disponibles, et auquel cas le BE peut demander à l'AAE de les construire (cf. section 4.1.2). Pour faciliter le travail, il est opportun de considérer la requête qui a échoué comme une connaissance partielle à ajouter dès à présent à la bibliothèque et pouvant être ultérieurement complétée lors de sa réutilisation par spécialisation, grâce à l'intervention d'un AAE compétent sur cette classe de problèmes.

4.2.4 Discussions

Dans l'approche décrite, la bibliothèque de niveau E1 est un simple fournisseur de styles évolution dont l'accès est explicitement demandé par l'utilisateur. Dès le départ, notre infrastructure n'a pas vocation à fournir un support à l'évolution automatique d'architectures et par conséquent ne cherche pas à assister et/ou guider la recherche et l'adaptation des styles compte tenu des besoins d'un AAE/DAE. Cette constatation peut être analysée plus finement selon les trois points suivants.

4.2.4.1 Approche statique

L'indexation et par conséquent la recherche de styles d'évolution n'est pas basée sur l'utilisation de mot-clés, mais sur leur organisation hiérarchique. Cette organisation a une influence directe sur la manière d'exploiter les styles d'évolution. Cette organisation prescrit entièrement le processus de recherche et d'intégration des styles. En effet, les relations de spécialisation et de composition définissent statiquement un chemin entre les styles d'évolution.

4.2.4.2 Niveau d'aide apportée

L'infrastructure n'intègre pas de fonctions d'adaptation et d'instanciation des styles. En effet, le style d'évolution réutilisé est "manuellement" adapté et/ou instancié par l'architecte dans l'application en cours d'évolution. Par ailleurs, l'infrastructure ne supporte pas une recherche de styles à partir d'un problème d'évolution donné à haut-niveau car la technique de recherche mise en place nécessite d'avoir déjà une "bonne idée" du style souhaité. En effet, l'infrastructure oblige l'utilisateur à décrire son problème sous forme d'un entête, ce qui nécessite a priori une connaissance précise du style recherché.

4.2.4.3 Niveau de réutilisation

Steven Wartik [WPD92] a proposé trois niveaux de maturité pour caractériser une approche par réutilisation : (1) le niveau de réutilisation ad-hoc, (2) le niveau de réutilisation opportuniste et (3) le niveau de réutilisation systématique. L'infrastructure que nous proposons se situe à un niveau 2, c'est-à-dire de réutilisation opportuniste. En effet, un architecte dispose d'une bibliothèque de styles réutilisables et de mécanismes pour intégrer et rechercher des styles. Cependant,

d'une part le producteur de styles d'évolution a la charge d'identifier des situations d'évolution pour lesquelles la réutilisation est possible et d'autre part, le consommateur de styles d'évolution a la charge de choisir les styles adaptés à son problème.

4.3 Réutilisation à travers l'instanciation

Nous venons de voir qu'un DAE peut rechercher des styles d'évolution correspondants à ses besoin en vue de les instancier sur son architecture nécessitant d'évoluer. Dans cette section, nous explicitons le fonctionnement de l'instanciation d'un type de style d'évolution et le processus d'évolution qui en découle. Dans SAEM, le processus d'évolution se réalise en quatre phases : une phase d'invocation, une phase de recherche (ou d'exploration), une phase d'exécution et enfin une phase de validation. Nous utilisons le diagramme d'états-transitions d'UML pour spécifier les différentes étapes de chaque phase. Le formalisme est décrit en Figure 4.8.

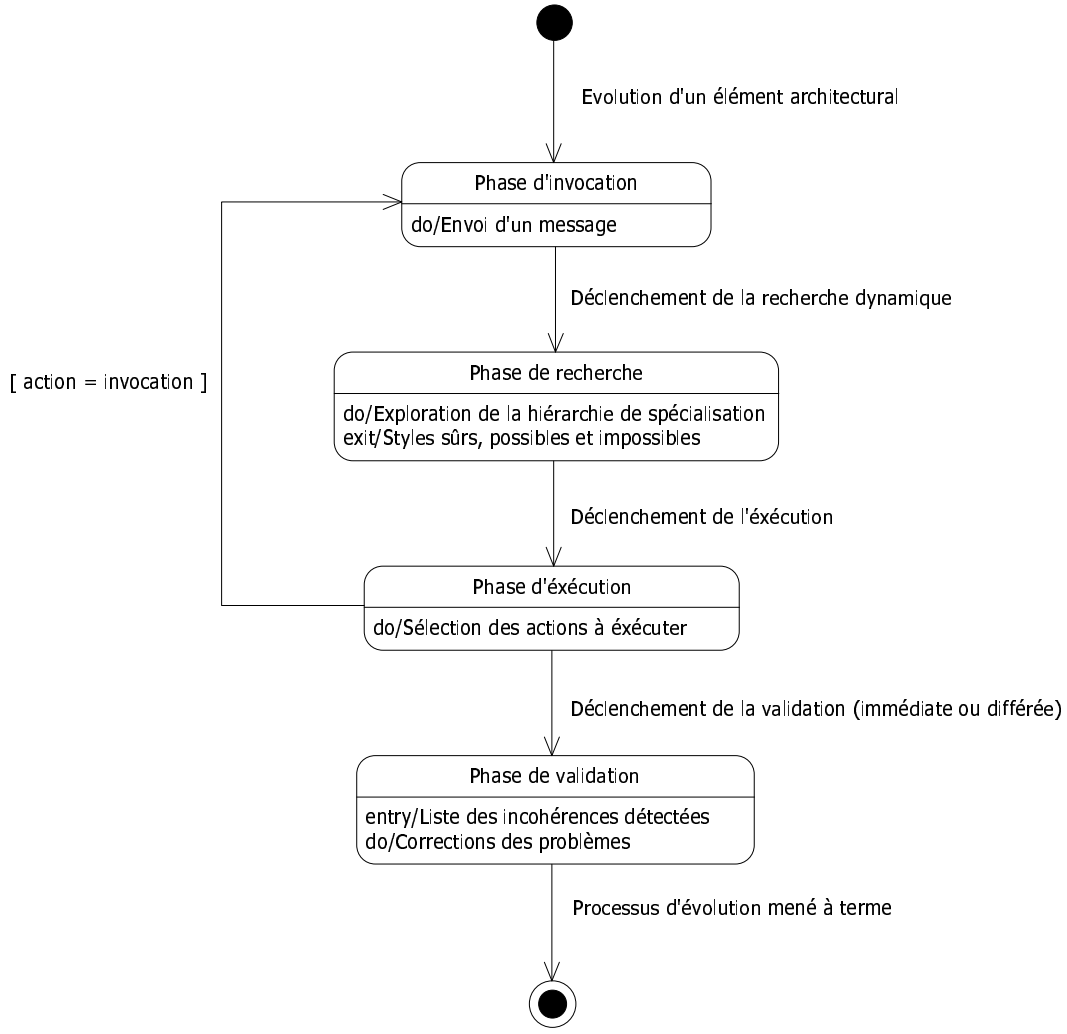

Figure 4.8 – Les quatre phases d'un processus d'évolution dans SAEM.

4.3.1 Phase d'invocation

Dans SAEM, l'instanciation et l'invocation d'un style sont confondus. Une invocation est un message à destination d'une instance en vue de déclencher l'exécution de la compétence implémentée dans son type de style d'appartenance (via le lien "est-un"). Un message peut être synchrone ou asynchrone selon le lien sémantique (spécialisation, composition ou utilisation) qui le véhicule. Le format standard d'un message est le suivant : `<invoke, ([paramètres]>`. A titre d'exemple, le message `<invoke, (server1,responseServer1)>` envoyé à une instance du style d'évolution `IncludePort@Component` (Figure 3.8) va exécuter la compétence pour permettre d'ajouter le port `responseServer1` à la structure du composant `server1`. Deux acteurs peuvent être à l'origine d'une invocation :

- Un DAE : pour réaliser une évolution, l'architecte choisi l'élément architectural qu'il souhaite faire évoluer ainsi que le style d'évolution à invoquer sur cet élément.
- Un style d'évolution : l'exécution de la partie compétence d'un style peut invoquer d'autres styles pour propager les impacts.

4.3.2 Phase de recherche

Lorsqu'un message est reçu par une instance et si la précondition définie dans la partie entête de son style d'appartenance est évaluée à vrai, alors ce dernier est *éligible*. Soit le style est concret et auquel cas l'exécution de sa compétence est déclenchée, soit le style est abstrait et auquel cas une recherche dynamique de compétence est lancée. Une fois de plus, la recherche dynamique de compétence repose sur un raisonnement classificatoire.

La classification permet ici de déterminer le style d'appartenance le plus spécifique de l'instance considérée dans la hiérarchie de spécialisation. Le raisonnement est guidé par la valeur des paramètres encapsulés dans le message. De ce fait, il est fréquent que les instances de styles à classer soit incomplètes. Pour traiter ce problème, nous nous inspirons des travaux sur la représentation des connaissances par objet, tel que SHIRKA [ER95], dans lequel a été proposée une classification tri-valuée, dite incertaine. En s'inspirant de cette classification, l'attachement d'une instance à un type de style d'évolution peut être qualifié de :

- *Sûr* si l'instance vérifie tous les contraintes (domaines de valeur des paramètres et précondition) du type de style.
- *Possible* si l'instance ne viole aucune des contraintes mais qu'il s'agit d'une instance incomplète.
- *Impossible* si l'instance en viole au moins une.

Un exemple schématique de la classification incertaine est présenté dans la Figure 4.9. Supposons l'invocation `oracle.S2()`, avec `oracle` un composant de type serveur, et *S2* un style d'évolution abstrait. L'instance réceptrice du message `<invoke, (oracle)>` est virtuellement issue du style abstrait *S2* ; le mécanisme de classification permet de déterminer le ou les sous-styles de *S2* auxquels cette instance peut être attachée. La valeur du paramètre `c` (abréviation du paramètre `context` pour des raisons de place) étant "oracle", les styles sûrs sont *S5*, *S8* et *S6*, car le type de leur paramètre `c` accepte la valeur "oracle". Toutefois, *S8* est un meilleur candidat que *S5* car il est plus spécifique, son type de paramètre étant `Server` alors que celui de *S5* est `Component` [5]. Le style impossible est *S7* car son type pour `c` n'accepte pas "oracle". Le

5. Cette technique demande de pouvoir inférer le type exact d'une instance (*e.g.*, `instanceof` en Java).

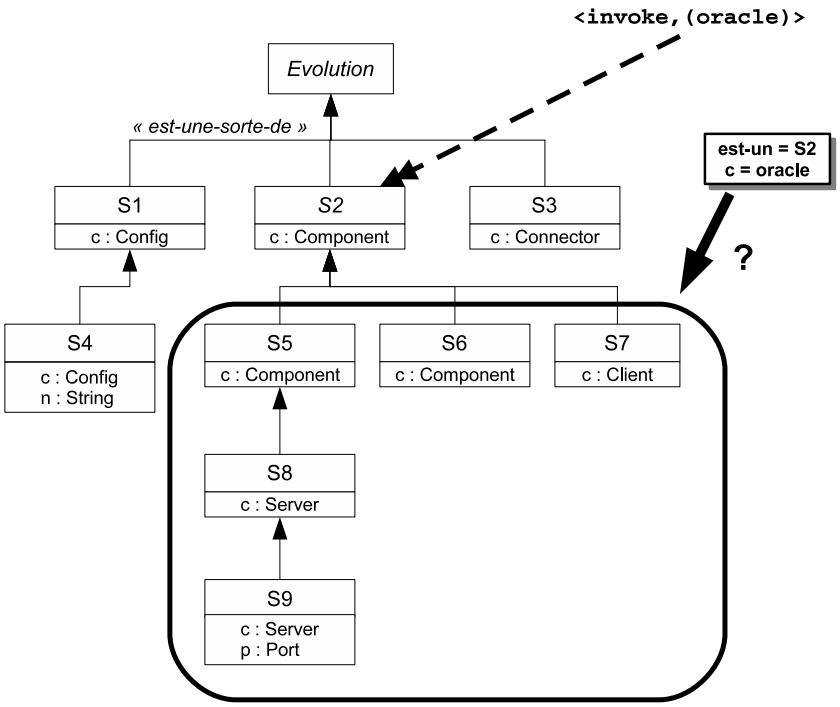

Figure 4.9 – Représentation schématique de la classification d'instance de style évolution.

style possible est *S9*, car le paramètre `c` ne suffit pas pour y classer cette instance. L'union des styles d'évolution sûrs et possibles forme l'ensemble E des styles éligibles. Plusieurs cas de figure se présentent alors :

- $card(E) > 1$: Le DAE doit intervenir pour le choix d'un style d'évolution à invoquer. Naturellement, les styles sûrs sont présentés prioritairement à l'architecte.
- $card(E) = 1$: Le style est automatiquement invoqué.
- $card(E) = 0$: L'évolution s'arrête et le DAE est averti.

Afin de permettre au DAE d'annuler son choix, et par ailleurs de lui permettre de sélectionner un autre style, l'ensemble des styles d'évolution éligibles est mémorisé jusqu'à ce que l'évolution entamée soit validée.

4.3.3 Phase d'exécution

Le déclenchement d'un style revient à l'exécution de sa partie compétence, séquentiellement, instruction par instruction. Si une instruction correspond à une invocation de style, un message est envoyé et une nouvelle phase de recherche est lancée. Le processus se réitère pour chaque style exécuté. L'exécution des styles est propagée par l'envoi de messages à travers les liens de composition et d'utilisation. La spécialisation propage également l'évolution lorsque la compétence d'un sous-style contient une invocation explicite à son super-style. Dans ce cas, l'impact "remonte" le long de la hiérarchie de spécialisation.

Normalement, les paramètres effectifs fournis lors de l'invocation correspondent aux paramètres formels définis dans l'entête d'un style d'évolution. Dans le cas où il manquerait des

paramètres effectifs, ces derniers sont demandés au DAE. Ceci est typiquement le cas dans le cadre de la classification tri-valuée présentée ci-avant, où les styles possibles nécessitent des paramètres qui n'ont pas pu être fournis lors de l'invocation.

Enfin, le graphe des instances des styles d'évolution mémorise leurs liens ainsi que les données ayant servi à leur exécution, jusqu'à la validation de l'évolution par le DAE. Un tel graphe peut être stocké dans la bibliothèque de niveau E0 pour une réutilisation ultérieure sur une architecture à l'identique.

4.3.4 Phase de validation

La vérification de la cohérence se fait au vu des invariants des styles d'évolution, qui ne sont autres que les contraintes structurelles posées sur les éléments évolutifs des styles exécutés. La validation dépend du type de traitement des incohérences choisi par l'architecte, et le cas échéant, la vérification peut être effectuée selon deux modes. Cette phase accepte l'intervention du DAE pour des actions curatives.

4.3.4.1 Type de traitement

Le type de traitement concerne la prise en compte des invariants associés à un élément évolutif et peut prendre trois valeurs : *strict*, *lâche* ou *ignoré*. Le type strict (par défaut) signifie que l'invariant doit être pris en compte dans l'évolution. Ce type de traitement assure la cohérence d'une évolution dans la mesure où l'élément architectural évolué et plus généralement l'architecture continue de satisfaire les choix de conception d'origine. A l'opposé, le type ignoré fait en sorte de ne pas tenir compte des invariants. Cela revient à considérer l'ensemble des éléments architecturaux comme non contraints, et que par conséquent l'évolution est "libre". Le type lâche est un traitement intermédiaire qui signifie qu'il est possible de violer certains invariants lors d'une évolution. Par conséquent, l'élément architectural évolué et plus généralement l'architecture peut diverger des choix de conception d'origine. On peut voir ces trois types de traitement comme des niveaux de sévérité différents portant sur le diagnostic des incohérences structurelles.

4.3.4.2 Mode d'évaluation

Dans le cas où le type de traitement est strict ou lâche, le mode d'évaluation choisi entre en jeu. Cet aspect est inspiré de la spécification du MOF [6]. Ce mode peut prendre deux valeurs : *immédiat* ou *différé*. Le mode immédiat signifie que l'invariant doit être vérifié à chaque modification de la structure de l'élément architectural contraint et plus généralement de l'architecture. Le mode différé signifie que la vérification de l'invariant pourra être déclenchée à n'importe quel moment, sur action de l'architecte par exemple. Pouvoir choisir entre ces deux modes nous paraît important. En effet, pour certains invariants, le mode différé s'avère mieux adapté. Un exemple est l'invariant qui valide la connexion entre deux composants. Sa vérification ne doit être réalisée que si les deux ports sont reliés par un connecteur.

Si aucune incohérence n'est détectée, alors l'évolution est validée et l'architecture est mise à jour. Dans le cas où des incohérences sont détectées, le DAE doit intervenir afin de corriger

6. Dans cette spécification, la classe *Constraint* possède un attribut *evalutionPolicy* qui explicite le mode d'évaluation de la contrainte

le ou les problèmes. La liste des invariants violés est présentée au DAE en vue de faciliter son travail. Ce dernier peut alors naviguer dans le graphe des instances des styles d'évolution et peut revenir à tout état antérieur, en annulant l'exécution de certains styles. Il peut aussi faire le choix d'autres styles en se basant sur la liste des styles éligibles, jusqu'à amener l'architecture à un état de cohérence structurelle.

4.4 Conclusion

Dans ce chapitre nous avons présenté une démarche pour guider les architectes vers la modélisation de leurs évolutions avec SAEM. Cette démarche se base sur le modèle MY introduit dans le Chapitre 3 et qui décrit les concepts du style d'évolution en utilisant trois branches : domaine, entête, compétence. En outre, ce modèle possède trois niveaux de modélisation et utilise des bibliothèques réutilisables pour chaque niveau. La démarche que nous proposons est consolidée par les différentes catégories d'acteurs que nous avons identifié, chacune impliquant un degré d'intervention et des responsabilités spécifiques.

Nous avons ensuite proposé une approche pour et par la réutilisation dédiée à la construction des bibliothèques d'éléments réutilisables pour chaque niveau. L'accent a été mis logiquement sur la construction de la bibliothèque de niveau E1, là où sont stockés les types de styles d'évolution. Notre infrastructure offre une opération d'intégration pour construire des hiérarchies cohérentes ainsi qu'une opération recherche pour y trouver les réponses à des problèmes d'évolution architecturale. Ces opérations sont nécessaires pour que les architectes puissent bénéficier des connaissances acquises par leurs pairs et surtout pour établir un système de réutilisation efficace. Pour ne pas introduire des nouvelles notions, notre infrastructure se contente d'exploiter la structure hiérarchisée de la bibliothèque à travers un raisonnement classificatoire.

Lorsque qu'un DAE décide d'instancier le type de style d'évolution de son choix sur les éléments de son architecture, un processus d'évolution est déclenché. La recherche dynamique de compétence intervient lorsque les liens entre les instances de styles d'évolution n'ont pas pu être déterminés statiquement en raison de l'utilisation de styles abstraits. Elle est un mécanisme d'inférence central dans notre infrastructure qui supporte un *polymorphisme de compétence*. La recherche dynamique de styles est un mécanisme de parcours de graphe descendant qui repose sur l'élément relationnel de spécialisation défini dans la section 3.4.2 et sur les conditions d'attachement d'une instance à un type de style d'évolution. Spécifiquement, une instance est attachée à un type de style si elle en vérifie la description de l'entête. Pour que la recherche dynamique soit un mécanisme entièrement automatique, c'est-à-dire ne nécessitant aucune intervention manuelle de la part du DAE, les instances à classer doivent être complètes.

A travers la phase de validation, notre système s'avère être un outil de simulation. En effet, tant que l'évolution n'a pas été explicitement validée, les modifications de l'architecture restent artificielles. Le graphe des instances des styles d'évolution offre un support de raisonnement au DAE, qui peut évaluer les répercussions d'une évolution sur le reste de son architecture et ainsi prendre sa décision en connaissance de cause.

Dans le dernier chapitre à suivre, nous procédons aux diverses réalisations et expérimentations que nous avons mises en place dans le cadre des idées développées dans cette thèse.

Chapitre 5

Réalisations et expérimentations

Au cours des deux chapitres précédents, nous avons expliqué notre modèle d'évolution SAEM et l'approche de réutilisation qui lui est associée. Il nous reste à proposer une réalisation possible et à mener une expérimentation autour de ces propositions. Ce chapitre est divisé en deux parties.

La première partie examine une stratégie de projection des concepts de SAEM vers ceux de l'ADL COSA. COSA est un représentant de l'école française puisqu'il a été développé par notre équipe dans le cadre d'une thèse de doctorat [Sme06]. La projection proposée a pour objectif de traduire SAEM et de le décrire en COSA. Comme il a été démontré qu'une description COSA peut être transformée en une description UML 2.0 [KSO06], SAEM bénéficie indirectement du formalisme UML 2.0 en vue de cibler des plateformes objets exécutables.

Dans la seconde partie de ce chapitre, nous menons une expérimentation autour de SAEM et de COSA dans le cadre d'un projet industriel nommé ZOOM. Ce projet cible les questions de conception et d'évolution architecturale appliquées à la modélisation de réseaux de tuyauterie pour la construction navale. D'une part, l'ADL COSA est exploité pour la description de tels réseaux sous formes d'architectures logicielles. D'autre part, SAEM a vocation à permettre à l'entreprise de capitaliser son savoir-faire d'évolution et de le ré-appliquer au moment où les réseaux modélisés sont amenés à évoluer. Un consultant expert métier a été désigné pour ce projet en vue de nous faire part de ses connaissances et de son expérience dans le domaine de la tuyauterie. En tant que chercheurs et informaticiens, notre travail a consisté à extraire les informations pertinentes pour les formaliser en exploitant les travaux de l'équipe MoDAL. Cette seconde partie représente une synthèse des différents livrables de MoDAL durant le projet ZOOM. Un prototype d'outil CASE a également été développé.

5.1 Projection de SAEM vers COSA

Dans cette section est considéré l'alignement de la hiérarchie de modélisation associée au paradigme des architectures logicielles en COSA (A2, A1, A0) avec celle associée au paradigme des styles d'évolution (E2, E1, E0). Pour ce faire, nous mettons en place une projection de SAEM vers COSA. Cette activité équivaut à la transformation d'un PIM (*Platform Independent Model*) en PSM (*Platform Specific Model*) dans le monde MDA®. Le choix de COSA est motivé par les raisons suivantes :

– COSA est un ADL qui emprunte les mécanismes opérationnels du paradigme objet, que l'on retrouve également dans SAEM. Ce n'est en réalité pas un hasard si COSA se révèle être une structure d'accueil idéale pour la description des styles d'évolution.

– Des stratégies sont proposées dans [Sme06] pour permettre le passage d'une spécification en COSA vers d'autres ADLs tels UML 2.0. UML constitue l'un des ADLs les plus utilisés dans la communauté industrielle. Ainsi, une spécification en COSA peut toujours être transformée en une spécification en UML 2.0. De ce fait, nous pouvons bénéficier de la gamme d'outils gravitant autour d'UML ainsi que des générateurs de code.
– COSA repose explicitement sur les trois niveaux de modélisation d'une architecture logicielle : le niveau méta-architecture, le niveau architecture et enfin le niveau application. Ceci facilitera notre tâche.

5.1.1 Présentation de COSA

Dans cette section, nous présentons COSA (Component-Object Software Architecture), un ADL hybride proposé par notre équipe [Sme06, KSO05]. Il se base sur la modélisation par objets et la modélisation par composants, pour la description des systèmes logiciels. COSA propose une description descendante de type "top-down". La contribution principale de cet ADL est, d'une part d'emprunter le formalisme des ADLs et de les étendre grâce aux concepts et mécanismes objets, et d'autre part d'expliciter les connecteurs en tant que citoyens de première classe pour traiter les dépendances complexes entre les composants.

5.1.1.1 Méta-modèle de COSA

L'ADL COSA décrit les systèmes en termes de classes et d'instances. Les types d'éléments architecturaux sont des classes qui peuvent être instanciées pour construire plusieurs applications. Les concepts de base d'une architecture COSA sont : le composant, le connecteur, la configuration, l'interface, les contraintes et les propriétés (fonctionnelles et non-fonctionnelles).

Le méta-modèle COSA est décrit en utilisant le diagramme de classe de la notation UML. Ainsi, les concepts de COSA sont représentés sous formes de classes, leurs caractéristiques sous formes d'attributs et les différents liens entre les concepts sont représentés par des associations. Par ailleurs, des classes abstraites qui n'ont pas de correspondances conceptuelles sont rajoutées uniquement comme support de factorisation, telles que les classes *ArchitecturalElementType* et *Interface*.

La Figure 5.1 décrit le méta-modèle COSA où certains détails ne sont pas représentés par soucis de clarté. Cette figure montre, en autres, que COSA sépare la notion de calcul (composant) de la notion d'interaction (connecteur) et distingue deux types d'interfaces : l'interface d'un composant appelé "port", et l'interface d'un connecteur appelé "rôle". Nous détaillons les principaux concepts de COSA dans les sections suivantes.

5.1.1.2 La configuration dans COSA

Dans COSA, la configuration est une classe qui peut être instanciée plusieurs fois et qui donnera plusieurs architectures d'un système. Une configuration peut avoir zéro ou plusieurs interfaces définissant les ports et les services de la configuration. Les ports sont des points de connexion qui seront reliés aux ports des composants internes ou aux ports des autres configurations. Les services représentent les services requis et les services fournis par la configuration. En général, les configurations sont structurées de manière hiérarchique : les composants et les connecteurs peuvent représenter des sous-configurations qui disposent de leurs propres architectures.

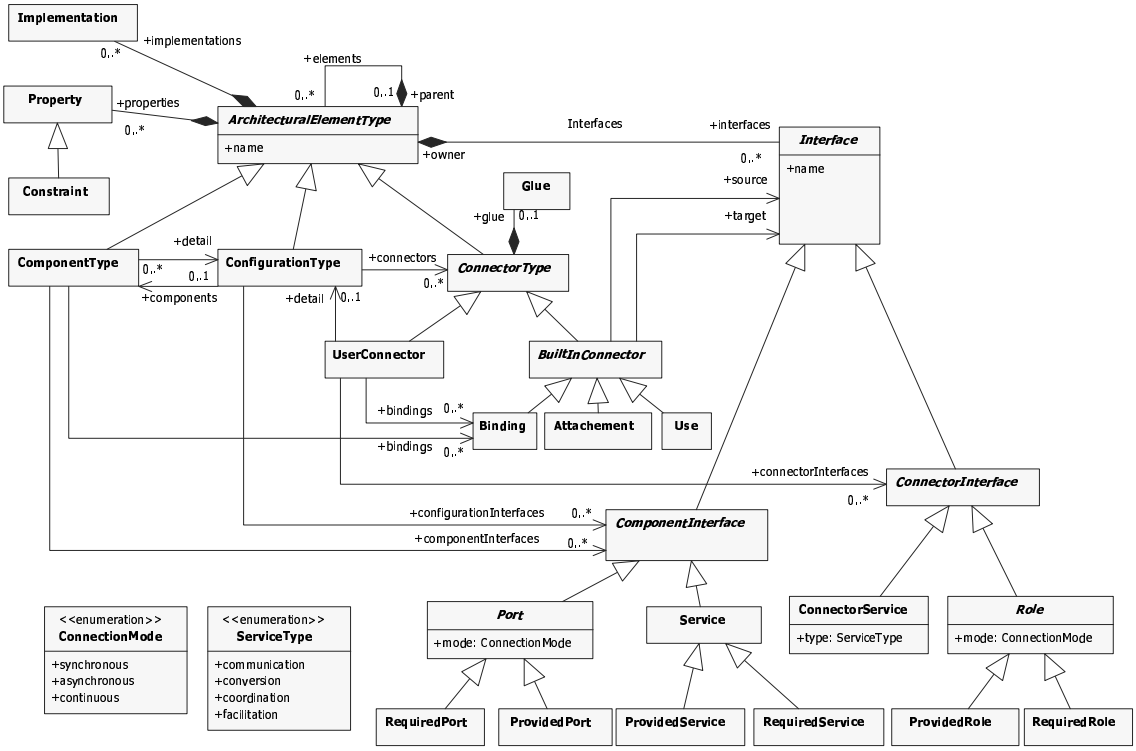

Figure 5.1 – Méta-modèle de COSA.

5.1.1.3 Le composant dans COSA

Un composant représente un élément de calcul et de stockage de données d'un système. Chaque composant possède une ou plusieurs interfaces. Un composant peut avoir plusieurs implémentations. Un composant peut être composite, dans ce cas il sera décrit par une configuration interne.

5.1.1.4 L'interface dans COSA

Dans COSA, les interfaces sont des entités abstraites de première classe. Une interface de COSA spécifie les points de connexion et les services fournis et requis pour un élément architectural (configuration, composant, connecteur). Ces derniers permettent à une interface d'interagir avec son environnement, y compris avec d'autres éléments.

- Le point de connexion : appelé également port pour les composants et les configurations, et rôle pour les connecteurs. Peut être soit un port fourni/rôle fourni, soit un port requis/rôle requis.
- Le service : décrit d'une part, le comportement fonctionnel de l'élément architectural en expliquant ce qu'il fait, on parlera alors de services fournis, et d'autre part, les fonctionnalités dont il a besoin pour fonctionner, il s'agit alors de services requis. Un service peut utiliser un ou plusieurs points de connexion pour exécuter sa tâche.

5.1.1.5 Le connecteur dans COSA

Dans COSA, les connecteurs sont définis comme des entités de première classe et sont utilisés pour connecter des composants, des configurations et des interfaces. COSA distingue deux catégories de connecteurs : les connecteurs utilisateur et les connecteurs de construction.

Connecteur utilisateur Un connecteur utilisateur est principalement défini par une interface et une glu. La glu décrit les fonctionnalités attendues d'un connecteur. Elle peut être un simple protocole reliant des rôles ou un protocole complexe ayant plusieurs opérations telles que la conversion de format des données échangées, leur transfert ou leur adaptation. Les connecteurs peuvent avoir leur propre architecture interne qui contient des calculs et du stockage de données. Dans le cas d'un connecteur composite, la configuration de ses sous-connecteurs et de ses sous-composants représentent la glu de ce connecteur.

Connecteur de construction COSA distingue trois sortes de connecteurs pour relier les interfaces des différents éléments architecturaux : attachement, binding, et utilisation.

- Attachement : une configuration en COSA est définie en énumérant un ensemble d'attachements qui lient les ports des composants ou d'une configuration aux rôles d'un connecteur. Un port requis peut être relié à un rôle fourni et un port fourni peut être relié à un rôle requis.
- Binding : quand un composant ou un connecteur et a fortiori une configuration disposent d'une description interne, une correspondance entre leurs descriptions externes et internes doit être mise en place. Le binding définit cette correspondance. Un binding fournit donc une association entre les ports (respectivement rôles) internes et les ports (respectivement rôles) externes des composants, des configurations et des connecteurs.
- Utilisation : le lien d'utilisation relie des services aux ports des composants ou aux rôles des connecteurs. Par exemple, un port fourni d'un composant est associé à un service fourni de ce composant et un port requis est associé à un service requis.

5.1.2 Les règles de passage des concepts de SAEM vers COSA

Nous suggérons de modéliser un type de style d'évolution suivant le patron de la Figure 5.2. Le patron considère deux points de vues. Le point de vue externe tout d'abord, qui considère un type de style d'évolution comme un type de composant fournissant un service d'évolution. Le point de vue interne ensuite, qui offre un niveau de détail plus poussé sur les éléments constitutifs d'un type de style d'évolution. Ces éléments constitutifs, nous les avons introduit avec la modélisation en Y dans le Chapitre 3. Pour rappel, il s'agit d'un type de domaine, d'un type d'entête et d'un type de compétence. La relation qui unit la vue externe et la vue interne d'un type de style d'évolution trouve son équivalent entre un type de composant (composite) et son type de configuration interne dans COSA.

Afin de pouvoir mettre en place ces correspondances entre SAEM et COSA, nous avons eu recours à l'augmentation du méta-modèle COSA.

5.1.2.1 Augmentation du méta-modèle COSA

COSA est un ADL "généraliste" qui peut être adapté à SAEM à travers la création d'une extension. Cette stratégie consiste à ajouter de nouveaux éléments de modélisations (de nouveaux

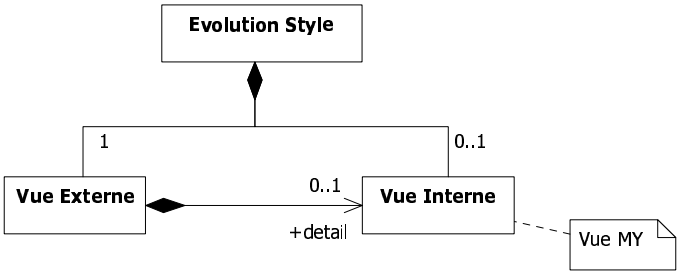

Figure 5.2 – Patron pour la modélisation d'un style d'évolution.

concepts) en modifiant directement le méta-modèle COSA. Ceci donne un nouveau langage de modélisation supportant d'une façon native les concepts de SAEM. La technique consiste à spécialiser les concepts de COSA pour introduire ceux exigés par SAEM et à spécifier formellement de nouvelles contraintes en OCL. L'augmentation du métamodèle COSA est illustrée en partie par la Figure 5.3, où les concepts originels de COSA sont représentés en grisés.

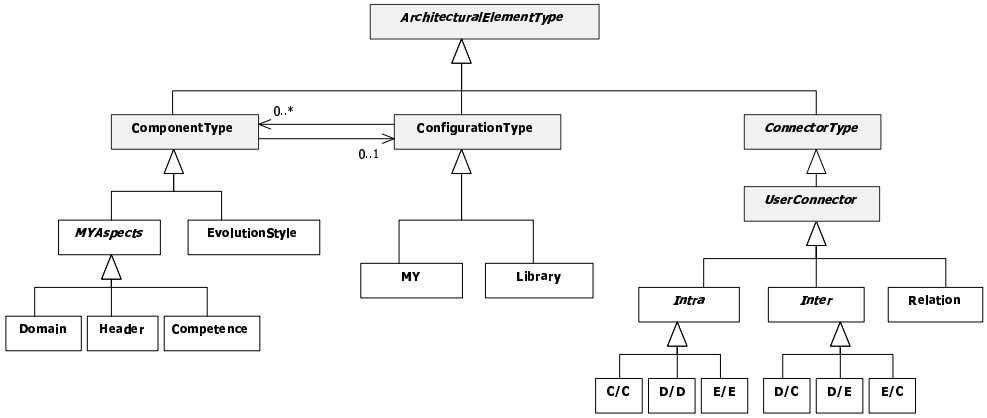

Figure 5.3 – Augmentation du méta-modèle COSA pour introduire la sémantique de SAEM.

On observe principalement qu'un type de style d'évolution est décrit à l'aide d'un type de composant COSA. Ce dernier est composite, car il possède sa propre configuration constituée de trois types de composants particuliers correspondants aux trois branches de MY. La bibliothèque de niveau E1 est quant à elle décrite à l'aide d'un type de configuration. Dans COSA, l'utilisateur peut définir ses propres types de connecteurs. Ce point fort de l'ADL est exploité pour décrire les différentes dépendances entre les éléments, aussi bien au niveau de la vue externe que de la vue interne. Les règles de passage les plus importantes entre les concepts de COSA et ceux de SAEM sont détaillées dans les sections suivantes.

5.1.2.2 Le style d'évolution (vue externe)

Le concept de style d'évolution, dans sa vue externe, correspond au concept de type de composant. Celui-ci peut être connecté à d'autres styles par l'intermédiaire de connecteurs de spécialisation, de composition et d'utilisation.
[1] Le concept de style d'évolution possède les types de ports requis *SourceSpecialization*, *Sour-*

ceComposition et *SourceUtilization*, et les types de ports fournis *TargetSpecialization*, *Target-Composition* et *TargetUtilization*

```
context COSA::SAEMextension::EvolutionStyle
inv : self.requiredPorts->forall(oclIsKindOf("SourceSpecialization") or
      oclIsKindOf("SourceComposition") or oclIsKindOf("SourceUtilization")) and
      self.providedPorts->forall(oclIsKindOf("TargetSpecialization") or
      oclIsKindOf("TargetComposition") or oclIsKindOf("TargetUtilization"))
```

[2] SEAM n'autorise que l'héritage simple. Ceci se fait en contrôlant qu'il n'existe au plus qu'un seul port requis de type *SourceSpecialization*

```
context COSA::SAEMextension::EvolutionStyle
inv : self.requiredPorts->select(oclIsKindOf("SourceSpecialization"))->size()<=1
```

[3] Un style d'évolution possède une vue interne décrite par un MY

```
context COSA::SAEMextension::EvolutionStyle
inv : self.detail->oclIsKindOf("MY");
```

5.1.2.3 Le style d'évolution (vue interne)

La vue interne d'un style d'évolution est décrite par un MY, qui correspond au concept de type de configuration de COSA. MY est un graphe formé de trois types de composant et de relations inter ou intra.
[1] Un MY ne contient au plus qu'un domaine, qu'un entête et qu'une compétence, ainsi que des relations inter et intra.

```
context COSA::SAEMextension::MY
inv : self.components->forall(oclIsKindOf("Domain") or oclIsKindOf("Header")
      or oclIsKindOf("Competence")) and
      self.connectors->forall(oclIsKindOf("Inter") or oclIsKindOf("Intra")) and
      self.components->select(oclIsKindOf("Domain"))<=1 and
      self.components->select(oclIsKindOf("Header"))<=1 and
      self.components->select(oclIsKindOf("Competence"))<=1
```

5.1.2.4 Spécialisation

Le concept de relation de spécialisation correspond au concept de connecteur utilisateur dans COSA. Une relation de spécialisation sert à relier deux styles d'évolution.
[1] *Specialization* hérite de *UserConnector*. *Specialization* possède strictement deux rôles en mode synchrone, dont l'un requis et l'autre fourni.

```
context COSA::SAEMextension::Specialization
inv : self.roles->size()=2 and self.requiredRole->exists(oclIsKindOf("to"))
      and self.providedRole->exists(oclIsKindOf("from")) and
      self.roles->forall(r | r.owner->oclIsKindOf("EvolutionStyle")
      and r.mode = #synchronous)
```

5.1.2.5 Composition

Le concept de relation de composition correspond au concept de connecteur utilisateur dans COSA. Une relation de composition sert à relier deux styles d'évolution.
[1] *Composition* hérite de *UserConnector*. *Composition* possède strictement deux rôles en mode synchrone, dont l'un requis et l'autre fourni.

```
context COSA::SAEMextension::Composition
inv : self.roles->size()=2 and and self.requiredRole->exists(oclIsKindOf("to"))
      and self.providedRole->exists(oclIsKindOf("from")) and
      self.roles->forall(r | r.owner->oclIsKindOf("EvolutionStyle")
      and r.mode = #synchronous)
```

5.1.2.6 Utilisation

Le concept de relation d'utilisation correspond au concept de connecteur utilisateur dans COSA. Une relation d'utilisation sert à relier deux styles d'évolution.
[1] *Utilization* hérite de *UserConnector*. *Utilization* possède strictement deux rôles en mode asynchrone, dont l'un requis et l'autre fourni.

```
context COSA::SAEMextension::Utilization
inv : self.roles->size()=2 and and self.requiredRole->exists(oclIsKindOf("to"))
      and self.providedRole->exists(oclIsKindOf("from"))
      self.roles->forall(r | r.owner->oclIsKindOf("EvolutionStyle")
      and r.mode = #asynchronous
```

5.1.2.7 La bibliothèque

La bibliothèque de SAEM est définie comme un type de configuration de COSA. Elle possède un seul type de port fourni.
[1] *Library* hérite de *ConfigurationType*. *Library* est un graphe de styles d'évolution et de relations, et a une seule interface fournie.

```
context COSA::SAEMextension::Library
inv : self.components->forall(oclIsKindOf("EvolutionStyle") and
      self.connectors->forall(oclIsKindOf("Relation") and
      self.interfaces->size()=1 and self.requiredPort->isEmpty()
```

[2] *Library* possède au moins deux styles d'évolution, l'un étant la racine de la hiérarchie de spécialisation, l'autre étant la racine de la hiérarchie de composition.

```
context COSA::SAEMextension::Library
inv : self.components->select(s | s.name="Evolution")->size()=1 and
      self.components->select(s | s.name="NEV")->size()=1
```

[3] Chaque style d'évolution possède un nom unique dans la bibliothèque

```
context COSA::SAEMextension::Library
inv : self.components->forall (s1, s2 | s1.name=s2.name implies s1=s2)
```

5.1.2.8 Exemple d'illustration

Un exemple illustratif est donné par le diagramme de structure composite d'UML 2.0 de la Figure 5.4, puis commenté.

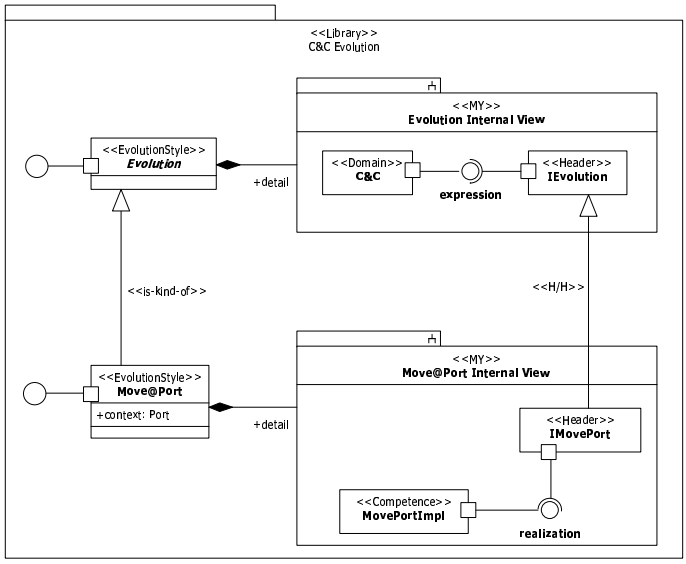

Figure 5.4 – Illustration d'une modélisation de styles d'évolution conformément au patron.

Prenons le cas du style d'évolution abstrait racine *Evolution*, subsumant l'ensemble des évolution dédiées aux architectures à base de composants. La vue interne de ce style dénote une modélisation en Y, composée du type de domaine *C&C* et du type d'entête *IEvolution*. Il s'agit ici d'une vue abstraite du MY, car le type de compétence n'est pas supporté. Repassons en vue externe, et intéressons-nous maintenant au style d'évolution *Move@Port*, sous-style de *Evolution*. Sa vue interne hérite du type de domaine *C&C*, et étend le type d'entête hérité avec son propre type d'entête *IMovePort*. On notera que la relation qui existe entre ces deux types d'entêtes correspond à une relation inter *H/H* définie dans l'extension du méta-modèle COSA. Enfin, la vue interne du style concret *Move@Port* contient le type de compétence *MovePortImpl*. De cette façon, une bibliothèque de styles d'évolution peut être construite à la manière d'une architecture logicielle à base de composants.

5.2 Le projet ZOOM

Le groupe STX Europe, spécialisé dans la construction navale, a lancé un projet dans le cadre du programme CAP-Excellence. Ce projet a pour objectif de répondre à un besoin d'intégration. L'intégration désigne la réalisation d'un système d'information intégré par la mise en relation (*i.e.*, interfaçage) de différentes applications existantes. L'interopérabilité, aussi bien sémantique que technique, est un enjeu crucial pour STX. On peut résumer la situation actuelle à un ensemble de tâches assistées par ordinateur, assurées par différents intervenants (Bureau d'étude, montage, achat, atelier, etc.), utilisant différentes applications et produisant des données de formats différents. Ce manque de cohérence entre les artefacts produits par les uns et les autres engendre des erreurs et nécessite une activité de synchronisation terriblement chronophage. Dès

lors, la problématique se pose en terme de communication, entre les acteurs d'une part, et entre les applications d'autre part. Le nom donné à cette ambition est la "CAO intégrée" ou "CAO unifiée".

Parmi les sous-projets que compte la CAO unifiée, on trouve le projet ZOOM, sur lequel l'équipe MoDAL est intervenue. Ce projet se focalise sur la modélisation de systèmes navals complexes.

5.2.1 Périmètre de notre intervention

Le projet ZOOM se propose d'explorer les moyens offerts pour modéliser les systèmes navals complexes. L'intervention de MoDAL se positionne sur deux points clés : l'organisation de la modélisation de ces systèmes selon plusieurs dimensions et la capitalisation du savoir-faire de l'entreprise pour faire face au départ des "sachants".

5.2.1.1 Des modèles à plusieurs vues et niveaux

A nouveau, nous exploitons le formalisme du modèle en Y. Cette fois-ci, il ne s'agit pas de décrire les concepts des styles d'évolution, mais ceux des architectures logicielles à base de composants [SOK08]. Spécifiquement, nous appliquons MY pour la modélisation des réseaux. Selon ce formalisme, l'architecture logicielle d'un système complexe peut être décrite par trois aspects (correspondant aux trois branches du Y) : configuration, composant et connecteur. MY supporte les trois niveaux de modélisation d'une architecture logicielle, à savoir le niveau méta-architecture, le niveau architecture et enfin le niveau application. La Figure 5.5 représente les réseaux d'un système naval à travers ses trois niveaux de modélisation, plusieurs de ses vues et selon un même niveau d'abstraction (diamètre du cercle).

Le cas d'un système naval – ou navire – est emblématique de la nécessité de disposer de plusieurs vues complémentaires afin de décrire le système dans son ensemble. Par exemple, les vues tuyauterie, gaine et électricité sont nécessaires à la description de l'aménagement des réseaux sur un navire. Elles correspondent basiquement aux différents corps de métier qui interviennent à bord. A ce propos, Desmond D'Souza [DSo01] parle de dimension horizontale d'un modèle.

D'autre part, il est important de faire varier le degré de finesse des informations à représenter à travers le niveau d'abstraction. On peut voir les différents niveaux d'abstraction comme des filtres destinés à ne retenir que les informations pertinentes pour un usage donné, reléguant ainsi les autres informations à des niveaux inférieurs. Il est important de rappeler que la décomposition hiérarchique offerte par les architectures logicielles, permettant de masquer les détails internes des composants notamment, est une technique d'abstraction phare. A ce propos, Desmond D'Souza [DSo01] parle de dimension verticale d'un modèle.

5.2.1.2 Capitalisation du savoir-faire

Dans le contexte de modélisation de systèmes navals, il est fait état d'un besoin crucial de capitaliser l'expertise des personnes pour faire face à leur départ de l'entreprise. C'est à ce niveau que MoDAL suggère l'emploi des styles d'évolution. Nous distinguons trois grandes catégories de styles : (a) les styles généralistes, (b) les styles métier et (c) les styles orientés projet. Ces trois catégories détermine chacune une portée de réutilisation des évolutions (cf. section 2.2.3.3).

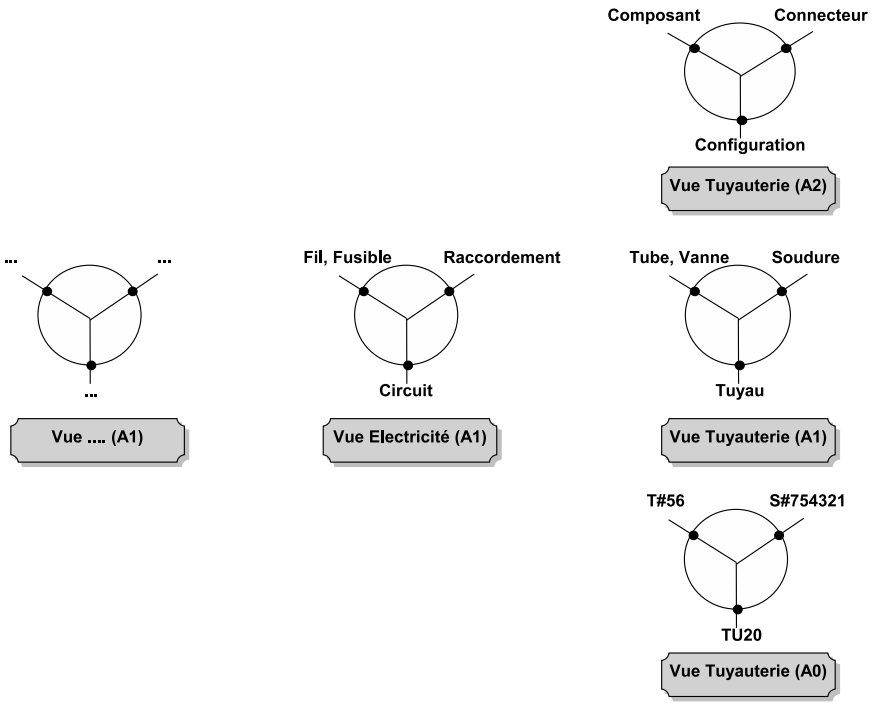

Figure 5.5 – Méta-modélisation en Y dans le cadre du projet ZOOM.

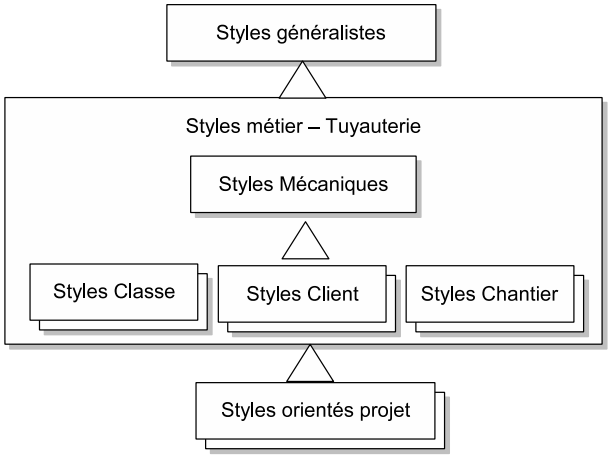

Figure 5.6 – Taxinomie de styles d'évolution dans ZOOM.

- Styles généralistes : ce sont des styles applicables pour tous les réseaux décris à base de composants et de connecteurs. Ce sont des styles fondamentaux dont le nombre est réduit et constant.
- Styles métiers : ce sont des styles applicables pour une famille de réseaux. En effet, pour chaque famille (réseau électrique, réseau de tuyauterie, réseau de gaines, etc.), on peut identifier un certain nombre de convenances. Ces styles capturent le savoir-faire des "sachants" dans l'entreprise et leur nombre peut être très important.
- Styles orienté projet : ce sont des styles spécifiques à un projet donné. Le nombre de ces styles n'est pas forcément important et répondent à des situations ponctuelles ou exceptionnelles, c'est-à-dire non anticipées.

Grâce à la connaissance du consultant expert métier, nous avons détaillé la taxinomie initiale, en décomposant le métier en une sous-taxinomie. Ainsi, il est possible d'identifier une hiérarchie plus détaillée de styles métier. Au plus haut niveau hiérarchique, nous trouvons les styles mécaniques. Ce sont des styles que l'on peut qualifier de "bon sens" pour faire évoluer un réseau de tuyauterie. Le recensement de ces styles tacites peut s'avérer difficile car ils sont enfouis dans la connaissance collective. Au niveau inférieur, nous trouvons les styles imposées par trois acteurs distincts : (a) les organismes de normalisation (appelés classes), (b) les clients (*i.e.*, les commanditaires) et (c) les chantiers navals. Ces trois acteurs œuvrent dans la même direction mais selon leur préoccupation respective : (a) la sécurité ; (b) la facilité de maintenance ou/et la cohérence par rapport à la flotte existante ; (c) l'optimisation des coûts et les ressources (humaines et matérielles) disponibles. Ainsi, chaque organisme de normalisation, chaque client et chaque chantier peut imposer son ensemble de styles d'évolution.

5.2.2 Analyse des besoins

La phase d'analyse des besoins vise à évaluer les attentes du projet ZOOM et leur adéquation avec les travaux de l'équipe MoDAL. Notre cas d'étude concerne exclusivement la vue tuyauterie (« TU » dans le jargon) des systèmes navals complexes. Dans ce cadre, nous sommes amenés à échanger fréquemment avec un consultant expert métier. Nous identifions des besoins en terme de conception et d'évolution d'architectures censées refléter des réseaux de tuyauterie.

5.2.2.1 Besoins relatifs à la conception d'architectures TU

Le point de départ a été d'apprécier dans quelle mesure les architectures à base de composants peuvent réellement supporter la description de réseaux de tuyauterie. Dans ce contexte, il est nécessaire d'identifier le mapping "one-to-one" entre les éléments architecturaux et les éléments physiques dans les navires (voir Figure 5.7).

Le domaine de la tuyauterie consiste à assembler des pièces manufacturées de différents types (tubes, coudes, vannes, etc.) pour l'écoulement de liquides (eau potable, eaux usées, eau de mer, etc.). En modélisation d'architecture, la frontière est parfois mince entre la responsabilité d'un composant et celle d'un connecteur. En règle général, le curseur de séparation s'appuie respectivement sur la distinction entre une abstraction de calcul et une abstraction de communication. A première vue, il semble raisonnable de modéliser les tubes par des connecteurs, de par leur rôle de médiums (ici de liquide). Or, dans un contexte non-logiciel comme c'est le cas ici, cette frontière est fragile et il s'est avéré plus cohérent de modéliser les tubes comme des composants,

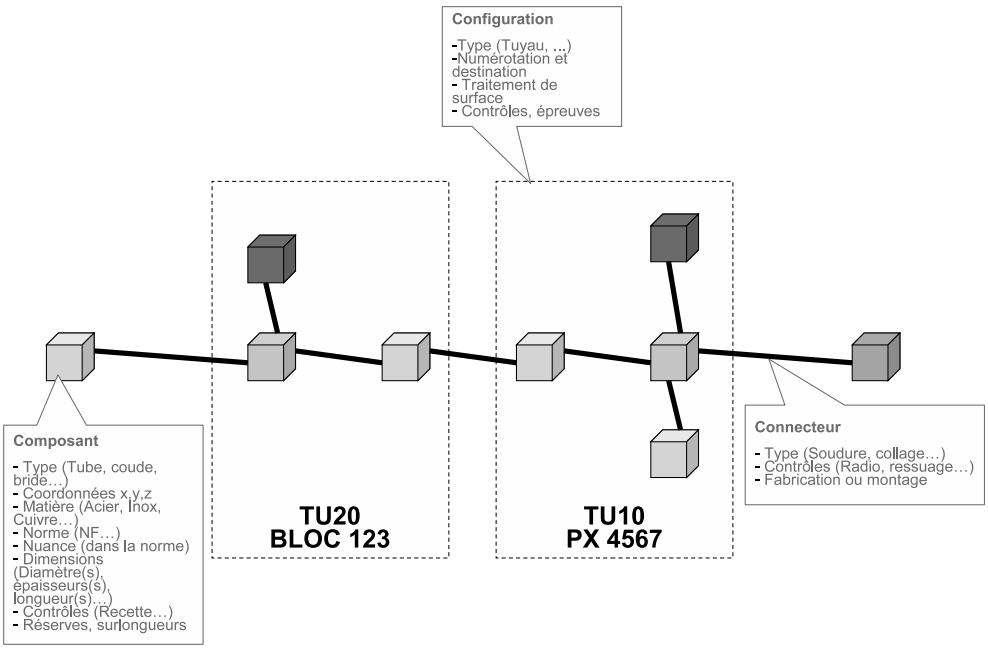

Figure 5.7 – Représentation schématique d'une modélisation de tuyauterie sous forme d'une architecture logicielle.

connectés entre eux par des éléments techniques de différentes natures (soudure, collage, etc.). Un assemblage de pièces manufacturées via des éléments techniques forme un tuyau, intégrable dans un réseau plus global. Par ailleurs, il est intéressant de remarquer à travers la Figure 5.7 que chacun des éléments de modélisation (composant, connecteur, configuration) présente un intérêt à être porteur de propriétés et donc à être réifié.

5.2.2.2 Besoins relatifs à l'évolution d'architectures TU

Il a été fait état par STX du besoin de faire évoluer ses réseaux de tuyauterie de façon incrémentale. Ces évolutions consistent principalement à compléter les réseaux par certains éléments et procéder au remplacement de certaines pièces. D'après l'expert, une action récurrente d'un opérateur CAO consiste à tronçonner les tubes de son réseau (cf. figure 5.8). Notamment, un tube doit être tronçonné lorsqu'il se situe à cheval sur deux blocs dans le navire. Dans ce cas, le tube d'un seul tenant doit être éclaté en deux tubes, dont la soudure devra être réalisée à bord du navire par un soudeur et non pas de manière préalable en atelier. Cette indication est portée par une propriété du type de connecteur soudure (voir propriété du connecteur de la Figure 5.7) pouvant être 'Fabrication' (*i.e.* soudure réalisée en atelier) ou 'Montage' (*i.e.* soudure réalisée à bord).

L'opération de tronçonnage est une évolution qui est actuellement réalisée manuellement par l'opérateur CAO. Seulement, tous les opérateurs ne respectent pas la même façon de procéder et/ou des erreurs peuvent être introduites. Bien souvent, ces erreurs ne seront détectées que plus tard, engendrant inévitablement des coûts pour les corriger. A ce titre, il est impératif de réutiliser des solutions éprouvées pour ce genre d'opération d'évolution.

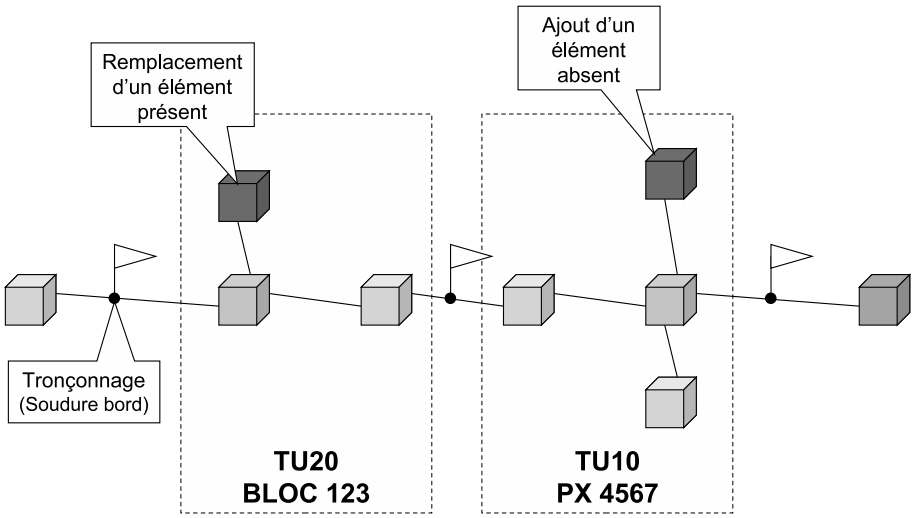

Figure 5.8 – Besoins d'évolutions d'une modélisation de tuyauterie sous forme d'une architecture logicielle.

A travers cette analyse des besoins, nous orientons nos propositions vers COSA pour les aspects liés à la conception et vers SAEM pour les aspects liés à l'évolution. Dans la section suivante, nous décrivons le travail réalisé dans le cadre du projet ZOOM sous forme d'une expérimentation de COSA et de SAEM pour la modélisation de réseaux de tuyauterie.

5.3 Travail réalisé au cours du projet ZOOM

Le défi du projet ZOOM réside dans la difficulté à extraire les informations pertinentes suite aux discussions avec le consultant expert métier. D'une part, un gros travail de spécification avec COSA est nécessaire avant de pouvoir aborder le problème de l'évolution avec SAEM. D'autre part, les savoir et savoir-faire d'évolution sont à la fois difficiles à identifier et difficiles à formaliser. Dans le même temps, nous devons être constamment force de proposition. Le travail réalisé inclut en parallèle le développement d'un prototype baptisé *COSAStudio* et dont les principales caractéristiques et fonctionnalités sont données en Annexe C.

5.3.1 Conception architecturale avec COSA

Cette section expose le travail qui a été réalisé concernant la modélisation des réseaux de tuyauteries à l'aide de l'ADL COSA. Le travail effectué au niveau A1 consiste à définir tous les types d'éléments architecturaux qui forment le vocabulaire métier du domaine de la tuyauterie ainsi que les possibilités de les assembler. Cet effort peut être appréhendé comme la définition d'un style architectural pour les systèmes de type *TU*. Le travail effectué au niveau A0 consiste à instancier les types d'éléments architecturaux pour décrire des réseaux de tuyauterie représentant le système à construire.

5.3.1.1 Niveau A1

Un style composant-connecteur (C&C) généraliste a été implémenté au sein du projet. En effet, nous partageons un point de vue qui consiste à considérer le paradigme composant-connecteur comme un style architectural [VCAO06]. Ce dernier indique que les éléments de l'architecture sont soit des composants, soit des connecteurs, ce qui fournit de fait un vocabulaire spécifique qui remplace la notion assez générique d'élément architectural. Ainsi, le style architectural C&C offre le vocabulaire de conception et les contraintes structurelles fondamentales pour la construction d'une architecture logicielle à base de composants. Les styles architecturaux "classiques" sont ensuite définis en extension du style C&C via la relation de spécialisation de COSA. Dans le contexte des systèmes navals, on peut trouver les styles Tuyauterie (TU), Electricité, Gaine, etc. Chacun fournit un vocabulaire dédié de conception et des contraintes structurelles spécifiques.

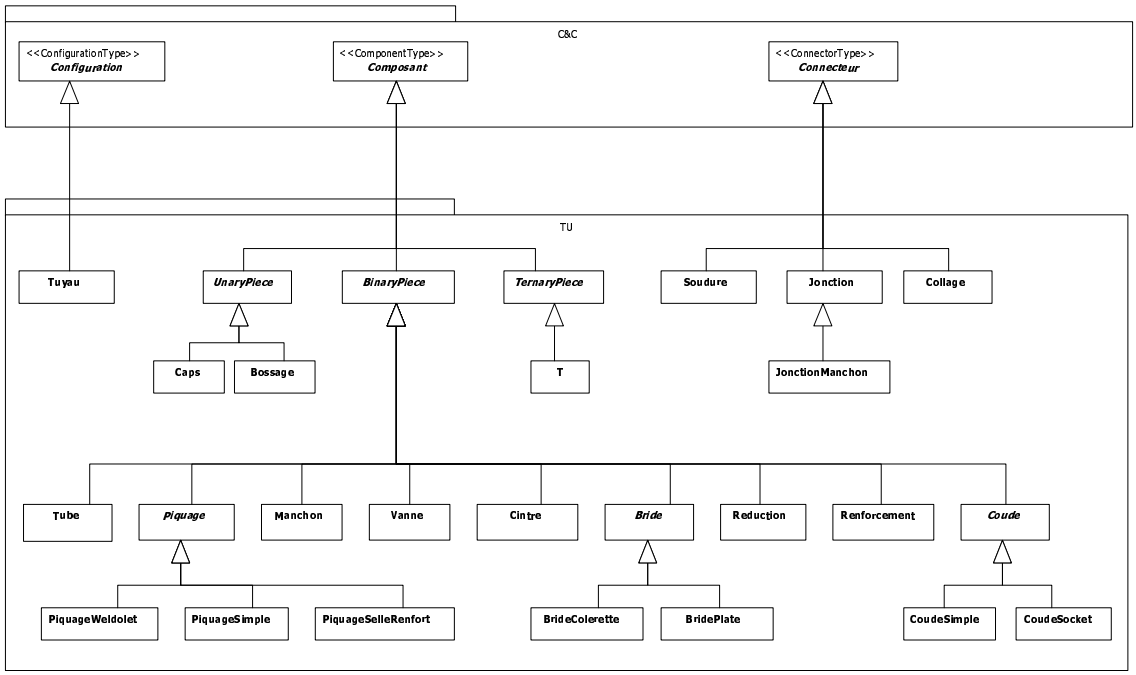

Figure 5.9 – Hiérarchie de types d'éléments architecturaux en COSA.

En interaction avec le consultant expert TU, nous avons recensé et hiérarchisé plusieurs types d'éléments architecturaux, que nous présentons sous forme d'un diagramme de classe UML à travers la Figure 5.9. La spécialisation a été principalement utilisée pour classer les différents types de pièces manufacturées et pour les distinguer par leur arité (unaire, binaire ou ternaire). Cette arité dépend à la fois du nombre de types d'interface différents qui ont été définis et du nombre d'instance maximale autorisée pour chaque type d'interface. Sans surprise, une large majorité des pièces sont binaires et par conséquent leurs éléments de raccord également. Les tubes par exemple, sont fondamentalement des pièces binaires (*i.e.*, une interface d'entrée et une interface de sortie), mais nous leur avons prévus des interfaces optionnelles pour y ajouter des piquages. Les piquages sont des tubes particuliers destinés à être connectés de manière transverse aux tubes après les avoir percés. L'intérêt de disposer d'interfaces explicites est évident ici et sert à guider le travail de l'opérateur CAO. Le type de connecteur jonction manchon est un exemple

de composite : il possède son propre type de configuration constitué d'un type de composant manchon et de deux types de connecteurs jonctions.

En plus des types d'éléments architecturaux spécifiés en COSA, nous leur associons un ensemble de contraintes structurelles. Dans COSA, les contraintes sont considérées comme des propriétés particulières et peuvent ainsi être portées par tout type d'élément architectural. A titre d'exemple, la table 5.1 répertorie certaines des contraintes définies sur les éléments de conception du style architectural C&C. La table 5.2 liste quelques-une des contraintes métiers que nous avons associé aux éléments de conception du style architectural TU.

Élément Architectural	Nom contrainte	Description
Connecteur	ConnectorValidity	Vérifie la validité des connecteurs architecturaux, c-a-d qu'ils possèdent au moins deux points de connexion valides
Composant et Connecteur	CardValidity	Vérifie que les composants et les connecteurs ont le bon nombre d'interfaces, comme cela a été spécifié par leur cardinalité au niveau de leurs types
Configuration	ArchStructureValidity	Détermine si la description contient assez d'éléments pour être la représentation d'un système (>= 1 composant ayant >=1 interface fournie, etc.)
Composant et Connecteur composite	Signature-Interface-MappingValidity	Vérifie que les composants et les connecteurs ayant des sous-architectures sont correctement connectés aux composants et connecteurs internes
Attachement	Interface-Direction-Validity	Vérifie que les directions des interfaces connectés sont compatibles (interfaces 'requises' se connectent uniquement aux interfaces 'fournies', etc.)
Configuration	Floating-Elements-Validity	Détecte les composants et les connecteurs qui ne sont pas connectés au reste de l'architecture
Configuration	DuplicatesValidity	Détecte les éléments avec des identifiants dupliqués

Table 5.1 – Exemples de contraintes structurelles définies dans C&C.

Élément Architectural	Nom contrainte	Description
Piquage	PiquageValidity	Vérifie que le diamètre du piquage est inférieur au diamètre du tube, selon un rapport d'au plus 2/3
Tube	PiquageOverhead	Détecte si un tube possède plus de 3 piquages, pouvant nuire à sa robustesse
Jonction	JonctionValidity	Vérifie que les pièces à joindre ont le même diamètre nominal
Soudure	SoudureValidity	Vérifie si la soudure est faite dans la même matière d'apport que les pièces afférentes

Table 5.2 – Exemples de contraintes structurelles définies dans TU.

5.3.1.2 Niveau A0

Le niveau application contient l'information "utile" à la construction des réseaux de tuyauterie du navire. Pour en donner une idée, nous choisissons un exemple didactique représenté par le schéma dans la partie gauche de la Figure 5.10. On y voit un ensemble de tubes (`tub1`, `tub2`

et `tub3`) joints à l'aide d'un coude simple `c1` et d'un T `t1`. Cet assemblage forme le tuyau `TU20`, qui possède trois points de jonctions possibles avec le reste du réseau de tuyauterie global.

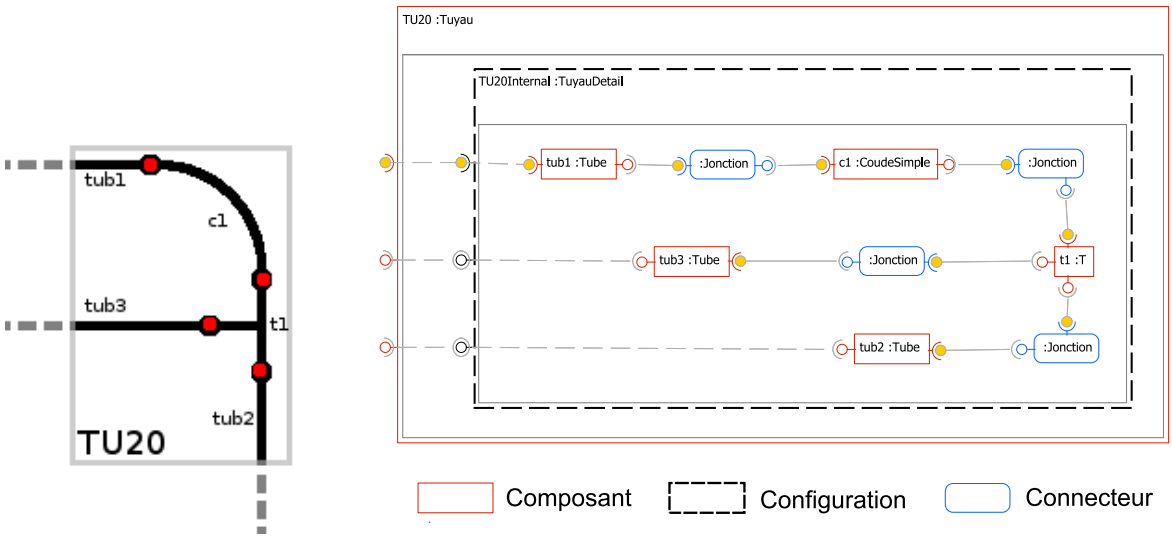

Figure 5.10 – Représentation schématique (à gauche) et représentation COSA (à droite) d'une portion de réseau de tuyauterie.

Ce même schéma peut être traduit en une application COSA, représentée dans la partie droite de la Figure 5.10. La notation graphique utilisée pour décrire les éléments de l'application est celle de notre outil *COSAStudio*. Cette notation repose sur une différenciation visuelle du composant et du connecteur et sur la représentation de l'interface dans le style « lollipop », popularisé par Microsoft mais également familier des utilisateurs d'UML. Les attachements sont représentés par des traits pleins tandis que les bindings sont représentés par des traits pointillés.

5.3.2 Évolution architecturale avec SAEM

Cette section expose le travail qui a été réalisé concernant l'évolution des réseaux de tuyauteries, en couplant SAEM à COSA. Seule l'évolution du niveau application A0 d'une architecture logicielle présente un intérêt immédiat pour STX. Pour cette expérimentation, nous proposons un scénario faisant intervenir l'architecte d'applications évolutives (AAE) et le développeur d'applications évolutives (DAE).

5.3.2.1 Niveau E1

Nous rappelons que pour pouvoir gérer l'évolution d'une architecture au niveau application A0, l'AAE doit spécifier les styles d'évolution associés à chaque élément architectural défini au niveau A1 et qui a servi à la description de l'application. Ces styles décrits ainsi au niveau architecture seront appliqués à l'évolution de tout élément d'une application créée à partir de ce niveau. Le fait d'avoir séparé la définition du style C&C et du style TU permet d'y associer des styles d'évolution ayant une portée différente. Les premiers capturent des évolutions généralistes, c'est-à-dire réutilisables dans n'importe quelle architecture. Les seconds capturent des évolutions métiers, c'est-à-dire réutilisables uniquement pour une familles de systèmes, en l'occurrence de

tuyauterie. En outre, les styles d'évolution métiers peuvent spécialiser les styles d'évolution généralistes. Sur le même principe, des styles orientés projet peuvent être obtenus par spécialisation des styles métiers. Cette technique permet de créer une taxinomie de styles d'évolution, très proche de celle proposée en Section 5.2.1.2.

Conformément à l'approche de réutilisation que nous avons développé au chapitre précédent, l'AAE a la charge d'identifier les évolutions candidates à la réutilisation. Clairement, le tronçonnage d'un tube en est une. La Figure 5.11 se concentre sur une partie du contenu de la bibliothèque de niveau E1. On remarque que le DAE a intégré à la bibliothèque le type de style d'évolution *Tronconner@Tube* en tant que sous style de *Split@Component*. En effet, le tronçonnage d'un tube est bien une sorte de partitionnement de composant. L'entête du super-style est surchargé et sa compétence est redéfinie afin de prendre en compte les spécificités du métier de la TU. Ainsi, le tronçonnage d'un tube doit produire deux tubes dont les longueurs, une fois additionnées, représentent la longueur du tube tronçonné. De plus, une soudure doit être nécessairement ajoutée. Cette dernière doit être réalisée dans la même matière d'apport que les tubes qu'elle connecte.

La précondition de *Tronconner@Tube* s'assure que le tube à tronçonner ne contient pas de piquages et que la longueur de coupe désirée est supérieure au tiers de la longueur du tube. La postcondition exige que le tube d'origine n'appartienne plus, temporairement ou définitivement, à l'architecture. Cette décision est à l'appréciation de l'architecte et n'est donc pas prescrite par le style d'évolution. Les relations de composition informent sur ce qui est jugé comme l'essence d'un tronçonnage par l'expert, à savoir l'ajout de tubes (deux, précisément) au tuyau ainsi que l'ajout de soudures (une seule, précisément) exclusivement. Ces stratégies d'ajout sont à leur tour perçues comme la composition de la création de l'élément (allocation) et de son inclusion dans la structure d'accueil. Les connexions et déconnexions des ports des tubes vont dépendre de la topologie des éléments en présence. A ce titre, les relations d'utilisation inter-styles sont préférées. Remarquons également l'introduction de styles abstraits *Weld@TubePort* et *Unweld@TubePort* comme éléments de factorisation. Leur intérêt est illustré dans la section suivante.

5.3.2.2 Niveau E0

Le DAE va consulter la bibliothèque de niveau E1 pour y récupérer un ou plusieurs styles d'évolution susceptibles de répondre à son besoin. On suppose qu'il cherche avec un objectif de partitionnement (*Splitting*) sur un élément de son architecture. C'est là tout l'intérêt de disposer d'une classification préliminaire des évolutions selon leurs stratégies (ajout, retrait, fusion, partitionnement, etc.) dans la hiérarchie de spécialisation. Ces styles abstraits sont autant de racines de substitution à la racine unique *Evolution*, qui permettent de guider efficacement la recherche. Dans l'exemple, cela permet à l'AAE de rechercher des compétences uniquement dans la famille des évolutions relatives au partitionnement plutôt que dans toutes les évolutions possibles. On suppose dans notre scénario que sa recherche abouti.

Le DAE sélectionne l'élément architectural à faire évoluer et le style à instancier sur ce dernier. Il sélectionne dans notre cas le tube `t2` et instancie le style concret *Tronçonner@Tube*. Dans notre scénario, `t2` ne possède aucun piquage et la longueur de coupe renseignée par l'architecte vaut 23,546. Les préconditions sont respectées et le style est éligible. Par conséquent, sa compétence est éxécutée. La figure 5.12 illustre un extrait des instances de styles d'évolution crées et leur liens. Le diagramme de communication [1] d'UML 2.0 est utilisée pour privilégier l'aspect spatial

1. Appelé diagramme de collaboration en UML 1.0

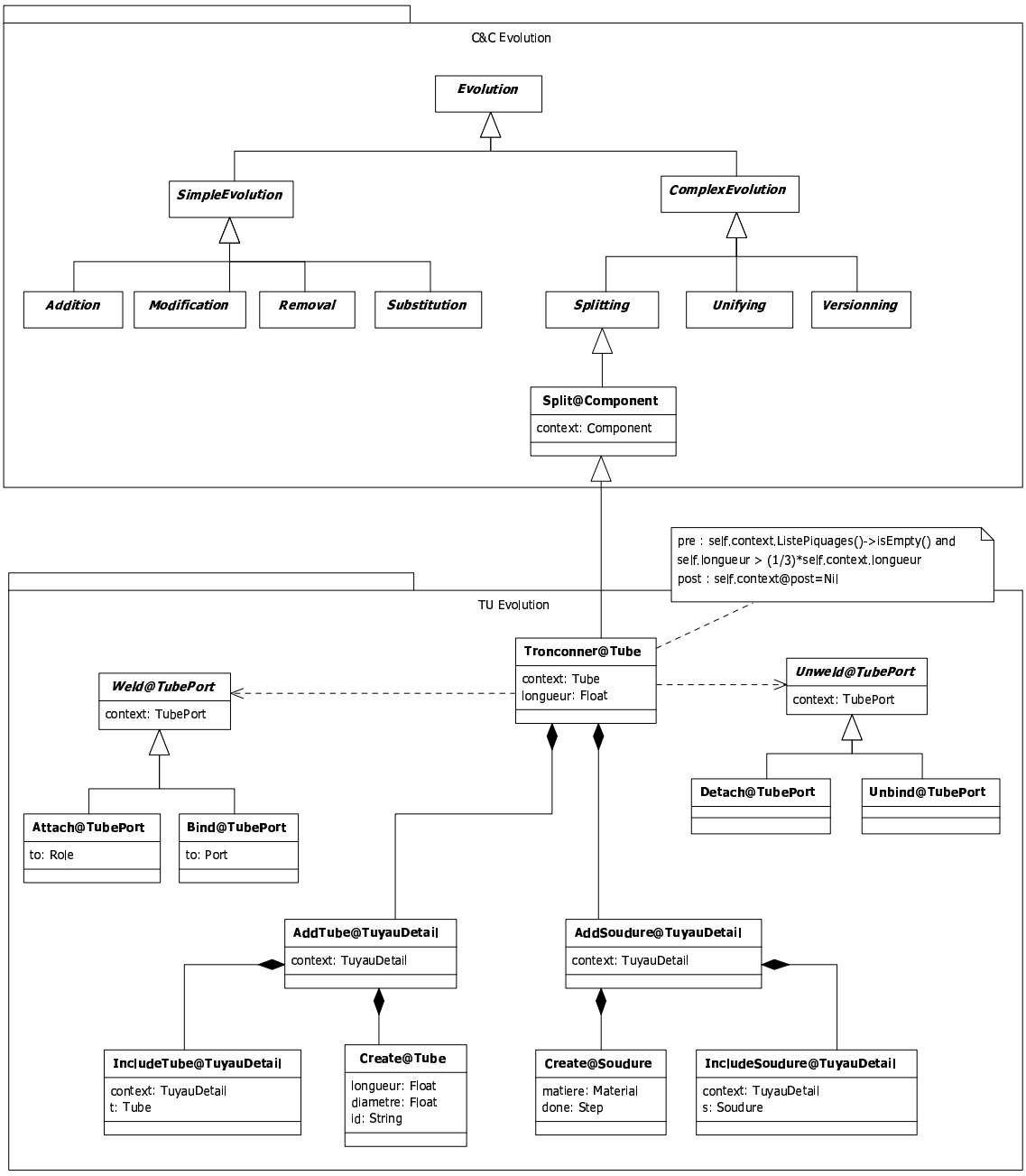

Figure 5.11 – La bibliothèque de niveau E1 (Vue externe des styles d'évolution).

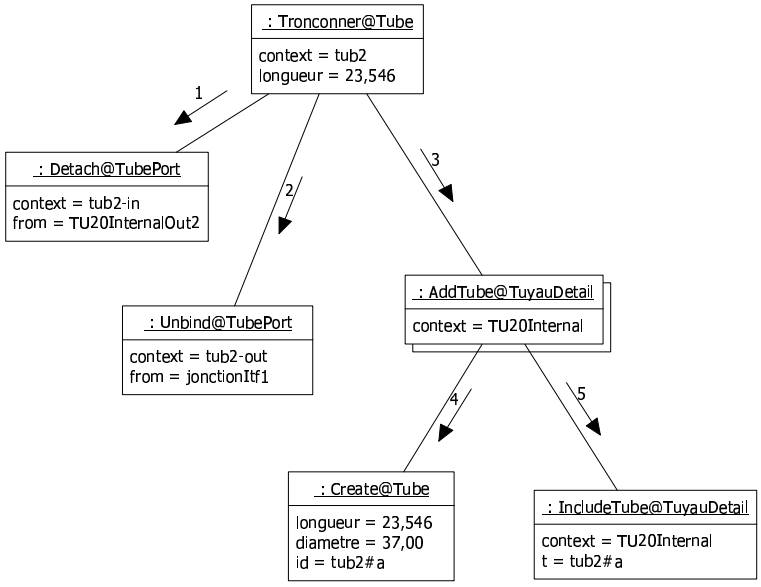

Figure 5.12 – La bibliothèque de niveau E0 (extrait).

des éléments du niveau E0 sur l'aspect temporel. La numérotation des invocations portés par les liens sémantiques donne une idée du flot des données spécifié par la compétence du style. Dans l'exemple, la compétence cherche d'abord à isoler le tube `tub2` à tronçonner en déconnectant ses deux ports `tub2-in` et `tub2-out` du reste de l'architecture. Dans le cas de `tub2-in`, il s'agit de supprimer un attachement, tandis que dans le cas de `tub2-out`, il s'agit de supprimer un binding. Pour ne pas présupposer de la nature des éléments à supprimer, la spécification de *Tronçonner@Tube* utilise le style abstrait *Unweld@TubePort*, laissant au mécanisme de recherche dynamique (cf. section 4.3.2) le soin de déterminer la bonne compétence à exécuter. Puis, la troisième étape de la compétence cherche à ajouter les deux nouveaux tubes `tub2#a` et `tub2#b` issus du tronçonnage du tube d'origine. L'exécution des compétences en cascade continue ainsi jusqu'à amener le processus d'évolution à terme. Le résultat attendu est celui décrit par la Figure 5.13.

En faisant le choix d'un type de traitement strict des invariants et d'un mode d'évaluation différé, aucune erreur ne devrait être diagnostiquée vis-à-vis des contraintes structurelles répertoriées dans les tables 5.1 et 5.2

5.4 Conclusion

Le lecteur peut s'étonner de trouver une chaîne de raffinement SAEM → COSA → UML à la place d'une projection directe de SAEM à UML 2.0 par l'intermédiaire d'un profil par exemple. Ce choix s'explique par le fait que nous ne considérons pas UML comme un vrai ADL, tandis que les ingrédients dont nous avons besoin pour décrire les styles d'évolution se trouvent dans l'ADL COSA. En effet, les concepts et les niveaux de modélisation de SAEM sont natifs et inhérents dans COSA. Indubitablement, l'environnement de développement UML (riche avec l'ensemble de ses outils) constitue le bon compromis de développement étant donné que le passage de COSA a UML a déjà été réalisé dans notre équipe. Néanmoins, il faut envisager le recours à l'intervention

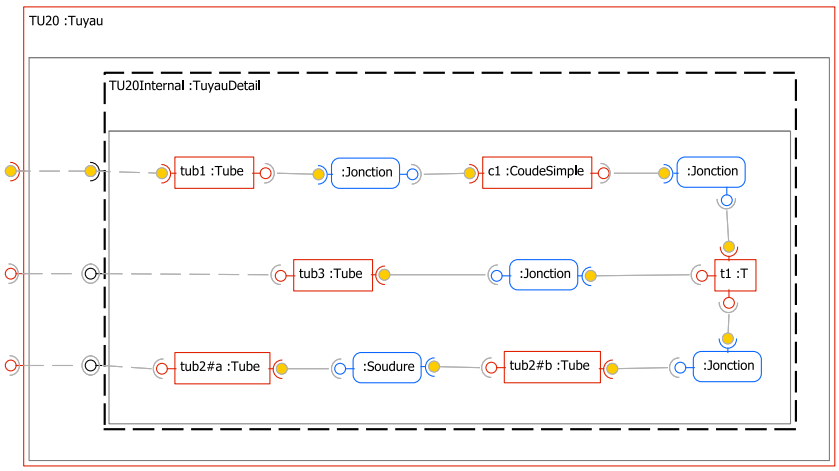

Figure 5.13 – Résultat du tronçonnage du tube `tub2`.

"manuelle" pour implémenter complètement les styles d'évolution. En effet, le degré de génération automatique du code atteint à l'heure actuelle est relativement faible.

Le modèle SAEM couplé à COSA permet d'aboutir à une approche pour la conception et l'évolution structurelle des architectures logicielles. COSA est un ADL qui nécessite de définir les types des éléments architecturaux avant de pouvoir les décrire en les instanciant. Aussi, l'étape de spécification réalisée au niveau A1 a été un préalable au travail mené avec SAEM. Le cadre du projet ZOOM a fourni à MoDAL un terrain d'expérimentation dont la particularité lui a permis de s'affranchir d'un contexte purement logiciel pour y trouver des correspondances tangibles avec les éléments montés à bord d'un navire. Cela montre que l'architecture logicielle à base de composants et la problématique de son évolution peuvent convenir à une variété de besoins.

Face à la difficulté à extraire les connaissances relatives à l'évolution, moins d'une dizaine de styles d'évolution ont été modélisés dans le cadre du projet ZOOM. En outre, ces styles encapsulent des "micro-évolutions". Le problème du passage à l'échelle de SAEM est un défi, surtout dans le cadre d'un projet industriel comme c'est le cas ici.

Conclusion et perspectives

L'évolution logicielle est une problématique qui présente une forte vitalité au sein la communauté de recherche. En témoigne le nombre d'ateliers et de conférences – au niveau national et international – dédiés à ce sujet qui n'a fait qu'augmenter ces dernières années. L'évolution logicielle est devenue une discipline majeure du génie logiciel, mais elle demeure complexe à traiter car profondément protéiforme. En effet, l'évolution intervient à toutes les étapes de la vie d'un système et à tout niveau d'abstraction et de granularité. En conséquence de cela, les pistes de recherche pour aborder cette problématique sont vastes. Face à ce constat, Mens *et al.* [MWD*05] ont tenté de lister les grands défis que doit relever la discipline de l'évolution logicielle. Parmi ceux-ci, on trouve la nécessité de disposer de nouvelles abstractions pour comprendre et maîtriser l'évolution dans les systèmes complexes. Nous pensons que le travail présenté dans ce manuscrit s'inscrit dans ce mouvement.

Dans cette thèse, nous avons circonscrit le vaste problème de l'évolution logicielle au domaine précis de l'évolution structurelle dans les architectures à base de composants, et basé toute notre recherche sur le postulat que la réutilisation était possible sur ce domaine. Analysons ce qui a été fait au niveau de ces travaux.

Bilan

Au Chapitre 1, nous avons évoqué plus de quinze ans de pratique dans le domaine des architectures logicielles et tenté d'éclaircir certains points. L'architecture logicielle est une discipline bâtie sur un principe d'abstraction qui s'avère particulièrement adaptée aux systèmes complexes. Le plus souvent, elle inclut une description des systèmes sous formes de composants et de connecteurs à gros grain, qui fournit un bon support de raisonnement. L'approche par composants se distingue de l'approche par objets grâce à une modélisation plus explicite et plus riche des systèmes, favorisant la réutilisation de leurs éléments. L'approche par composants est souvent perçue comme une technologie "post-objet" qui ambitionne le passage de l'ère artisanale à l'ère industrielle dans le domaine de l'ingénierie du logiciel. Ainsi, des langages spécifiques à l'approche composant ont vus le jour, ce sont les ADLs. Nous avons présenté un vue d'ensemble des ADLs proposés dans la littérature. Enfin, nous avons considéré la notion d'architecture logicielle comme concept unifiant des systèmes complexes, à l'instar de la notion de modèle considérée par l'OMG, arguant différents niveaux de modélisation (A3, A2, A1 et A0).

Notre première contribution est liée à l'état de l'art sur les modèles d'évolution structurelle des architectures logicielles autour des ADLs. Ainsi, nous avons proposé dans le Chapitre 2 un cadre de comparaison de l'évolution architecturale sous la forme d'une pyramide des besoins à trois niveaux : l'intention du modèle vis-à-vis des grandes classes d'objectifs pour l'évolution, le pouvoir expressif du modèle et enfin sa qualité. Nous avons mis en évidence que tout modèle d'évolution ou toute problématique d'évolution peut être positionnée dans cette pyramide. Ainsi,

nous avons mené une étude détaillée des travaux dans le domaine de l'évolution architecturale en positionnant chaque modèle d'évolution rencontré dans la pyramide des besoins. Ce travail nous a permis de dégager les aspects de l'évolution architecturale qui ont été largement traités, peu traités ou jamais traités. Il s'avère que l'aspect réutilisation n'a été que faiblement traité par les modèles d'évolution existants et qu'à ce titre, la proposition d'un nouveau modèle d'évolution centré sur cet aspect présente un fort intérêt.

Au Chapitre 3, nous avons discuté de la nécessité de séparer l'évolution des données et des traitements au sein de la description d'un élément architectural. Ainsi isolée, l'évolution peut être durablement capitalisée et réutilisée. Puis, nous avons argumenté la pertinence du concept de style pour remplir cet objectif. Partant de ces réflexions préliminaires, nous avons introduit notre modèle d'évolution à base de style, baptisé SAEM. L'utilisation de styles pour capturer des savoir et savoir-faire pour l'évolution architecturale nous semble une technique particulièrement prometteuse. Elle permet la formalisation des évolutions récurrentes en fonction des buts à atteindre et nécessite l'expression des relations inter-styles exprimant par exemple une dépendance dans leur utilisation. En outre, la technique de méta-modélisation en Y offre un formalisme de représentation simple et minimal pour le paradigme des styles d'évolution. Cette proposition peut être rapprochée des travaux entrepris dans UPML [FMvH*03] (*Unified Problem-solving Method description Language*) en particulier, où l'objectif reste d'obtenir une application de système à base de connaissance à partir de blocs réutilisables.

Un des objectifs fondamentaux du génie logiciel consiste à réduire les efforts et le coûts liés au développement et la maintenance des systèmes. Comme il est plus efficace de rependre des artefacts existants que de "réinventer la roue" à chaque fois, le génie logiciel a naturellement intégré ce principe sous la forme de deux motivations inter-dépendantes : le développement pour et par la réutilisation. A travers le Chapitre 4, nous avons cherché à récupérer ce principe important pour l'appliquer à l'évolution des architectures logicielles. Dans ce but, nous avons proposé une démarche pour guider l'architecte dans son processus de modélisation d'évolution logicielle. Cette démarche améliore la réutilisation des styles d'évolution en supportant des bibliothèques multi-abstractions et multi-vues pour les trois niveaux de modélisation du paradigme des styles d'évolution (E2, E1 et E0). Une infrastructure a été bâtie autour des bibliothèques de styles pour prendre en charge leur peuplement et leur interrogation. Le cœur de ces deux fonctionnalités repose sur un raisonnement classificatoire. Nous pensons que cette infrastructure a le mérite de ne pas introduire de nouveaux concepts par dessus SAEM afin de raccourcir la courbe d'apprentissage de l'architecte.

Le dernier chapitre a cherché à poursuivre les investigations autour de SAEM ainsi qu'a expérimenter nos propositions. La première partie de ce chapitre a défini la projection de SAEM vers COSA, un ADL hybride développé par notre équipe. L'idée de projeter les concepts de SAEM vers COSA permet d'enrichir ce dernier de charges sémantiques propres aux styles d'évolution mais aussi de profiter de la traduction de COSA à UML en vue de produire une implémentation. Le projet ZOOM a montré que les problématiques de conception et d'évolution architecturale existent, pour tout type de système. Les idées développées dans cette thèse ont été exploitées afin de répondre à un besoin de capitalisation d'expérience auquel fait face l'entreprise STX. Reste l'épineux problème du passage à l'échelle de SAEM. De plus amples retours d'expérience nous permettront d'étudier de manière concrète les limites de nos propositions.

Mais ce travail n'est pas clos pour autant, des approfondissements restent à faire. Voyons ce qu'il en est.

Perspectives

Le travail de cette thèse a permis la mise en place d'un cadre conceptuel pour la modélisation de l'évolution à l'échelle de l'architecture des systèmes, fortement teinté de réutilisation. En outre, cette thèse a cherché à dépasser le problème de l'évolution architecturale pour le placer dans une perspective plus générale d'ingénierie des connaissances. Le travail réalisé peut se prolonger vers huit perspectives de recherche, triées par échéance décroissante.

Vers un modèle d'évolution réflexif

L'idée est de rendre le SAEM réflexif, c'est-à-dire d'appliquer SAEM à lui même afin de le rendre évolutif, adaptable et ouvert. A notre connaissance, peu de modèles d'évolution peuvent prétendre à des qualités d'auto-évolution comme celles-là. Pour ce faire, il est nécessaire de considérer les concepts définis dans le méta-modèle de SAEM comme des éléments évolutifs.

Styles d'ordre supérieur et styles de co-évolution

Premièrement, la projection de SAEM vers COSA a préparé le terrain pour l'émergence de styles d'évolution d'ordre supérieur. En effet, à partir du moment ou les styles sont décrits à l'aide d'éléments d'un ADL, ils peuvent évoluer à travers des styles d'évolution, au même titre que n'importe quelle autre architecture. Ces styles d'évolution appliqués à eux-mêmes sont appelés styles d'évolution d'ordre supérieur. A titre d'exemple, l'opération d'intégration proposée par l'infrastructure de réutilisation pourrait être spécifiée comme un style d'ordre supérieur. Deuxièmement, il est important de rappeler que les styles d'évolution, qu'ils ciblent le niveau architecture (A1) ou bien application (A0), sont stockés de manière uniforme dans la bibliothèque de niveau E1. Une nouvelle gamme de styles peut être obtenue par composition de styles ciblant différents niveaux de modélisation, et appelés styles de co-évolution. Ces derniers doivent permettre de maintenir le lien de causalité entre les niveaux de modélisation d'une architecture au cours de son évolution, en propageant les impacts d'une évolution du niveau A1 vers le niveau A0, et vice-versa.

Propagation d'impacts dirigée par la sémantique

En l'état actuel, l'AAE a la charge de spécifier des styles d'évolution ainsi que leurs relations sémantiques pour gérer la propagation des impacts. En effet, nous avons fait l'hypothèse que cela rentre dans le cadre de sa compétence. Pour automatiser la propagation, il est possible d'introduire dans l'architecture des propriétés sémantiques exprimant le degré de corrélation entre les éléments architecturaux. Cette idée a été utilisée par le passé pour la propagation du versionnement dans les objets [UO96]. L'idée ici serait de généraliser cette approche pour n'importe quel élément architectural et n'importe quel opérateur (ajout, suppression, fusion, etc.). De cette façon, l'effort de spécification de l'AAE serait réduit, reléguant une large partie du travail à l'exécution.

Évolution comportementale

Nous avons noté au début du Chapitre 3 que l'évolution était une composante qui pouvait concerner aussi bien la structure que le comportement des éléments architecturaux. Nous avons considéré uniquement l'aspect structurel dans nos propositions. Nous considérons que l'intégration de l'aspect comportemental dans notre travail actuel ne sera que complémentaire pour le modèle d'évolution que nous proposons.

Raffinement et versionnement

Les problèmes de raffinement et de versionnement des styles d'évolution n'ont pas été abordés dans le cadre de cette thèse. Nous suggérons un raffinement sélectif dans lequel seule la compétence d'un style d'évolution est raffinée à chaque étape, pour la même entête (préservant ainsi les "propriétés" de l'évolution décrite). Cette technique nécessite d'introduire une nouvelle relation sémantique de raffinement. Dans le même ordre d'idée, le versionnement peut être adressé par l'introduction d'une relation sémantique de dérivation entre les styles. Ainsi, il sera possible de faire co-exister dans la bibliothèque plusieurs versions et plusieurs raffinements d'un style d'évolution.

Styles d'évolution génériques

Bien que la spécialisation, la composition et l'utilisation permettent de réutiliser les spécifications, cela ne permet pas de résoudre tous les besoins de réutilisation. Ainsi, le SAEM pourrait intégrer un mécanisme de généricité. La généricité se rapporterait à la capacité de paramétrer des styles. Les styles d'évolution génériques s'inspirent des "templates" en C++ ou des types génériques en Eiffel par exemple. En effet, l'instanciation ne fournit pas de services pour paramétrer des styles. Or, souvent, des structures communes dans la description des évolutions dans les architectures sont amenées à être spécifiées à plusieurs reprises. Les styles d'évolution génériques permettent de réutiliser une spécification en fournissant au compilateur la manière de substituer le nom des types dans l'entête et la compétence. Ils constituent donc une description partielle des propriétés et des relations d'un ensemble de styles qui a l'avantage d'être adaptable à d'autres ensembles de styles pris dans d'autres situations.

Raisonnement par analogie

Une hypothèse (bien prouvée) est qu'il y a des problèmes et des solutions d'évolution qui apparaissent dans des domaines très différents mais qui se ressemblent. Dans la correspondance analogique, deux styles d'évolution sont déclarés similaires si ils sont deux instances d'une même structure générique. Le concept de style générique mentionné ci-avant tient ici naturellement le rôle de cette structure. Or, il est possible d'adapter l'algorithme de classification pour le faire fonctionner sur des styles génériques. Le résultat, c'est un raisonnement par analogie, ayant notamment pour objectif de réutiliser des styles d'évolution spécifiés initialement pour d'autres domaines que pour le domaine où le problème d'évolution s'est posé.

Au delà des règles : les styles

Dans cette thèse, les styles ont été utilisés dans le cadre de l'évolution logicielle. Pourtant, il est possible de généraliser ce travail à toute approche de type prescriptive. Les styles offrent un formalisme de type entête/compétence qui fait écho aux règles exprimées classiquement sous la forme prémisse-conclusion. Les modes de raisonnement mis en jeu divergent : le style repose sur un raisonnement classificatoire tandis que la règle repose sur un raisonnement déductif. D'autre part, les travaux intensifs menés sur les formalismes à base de règles ont montré leurs limites en termes d'abstraction, de structuration et de réutilisation. Clairement, le style est décrit à plus haut-niveau d'abstraction que la règle, et bénéficie d'une sémantique plus riche. A ce titre, une règle sémantique peut être perçue comme un style "atrophié".

Émulation

Un fait marquant est la convergence qui est apparue dans les derniers mois entre la thématique de cette thèse et le projet de David Garlan et son équipe sur les "Evolution Styles" [Gar08, CDPG*09]. Cependant, si l'on reconsidère la base de la pyramide des besoins, l'intention de leur modèle d'évolution à base de style diffère du nôtre. En effet, tandis que notre modèle d'évolution s'intéresse à la formulation de l'évolution et à ses impacts, leur proposition cherche à garder trace de l'évolution. Garlan *et al.* souhaitent modéliser les trajectoires évolutives d'une architecture comme une séquence d'étapes, chacune décrite par un opérateur d'évolution. Une trajectoire évolutive débute et se termine sur des configurations connues a priori, et passe par une série de configurations intermédiaires. L'approche proposée vise ainsi à explorer et à évaluer objectivement les différentes trajectoires évolutives d'un système.

Bibliographie

[AAG95] Abowd G. D., Allen R., Garlan D. : Formalizing style to understand descriptions of software architecture. *ACM Trans. Softw. Eng. Methodol. 4*, 4 (1995), 319–364.

[ADG97] Allen R., Douence R., Garlan D. : Specifying dynamism in software architectures. In *In Proceedings of the Workshop on Foundations of Component-Based Systems* (1997), pp. 11–22.

[ADG98] Allen R. J., Douence R., Garlan D. : Specifying and analyzing dynamic software architectures. In *Fundamental Approaches to Software Engineering* (March 1998).

[AFLS98] Angele J., Fensel D., Landes D., Studer R. : Developing knowledge-based systems with mike. *Automated Software Engg. 5*, 4 (1998), 389–418.

[All97] Allen R. : *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[And99] Andersson J. : Dimensions of dynamism. In *Proceedings of the Second Nordic Workshop on Software Architecture (NOSA'99)* (August 1999).

[AZ05] Avgeriou P., Zdun U. : Architectural patterns revisited - a pattern language. In *Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005)* (Irsee, Germany, July 2005).

[Bas98] Basson H. : An integrated model for impact analysis of software change. In *International Conference on Software Quality Management* (Amsterdam, Netherland, April 1998), Springer-Verlag Ed.

[BCDW04] Bradbury J. S., Cordy J. R., Dingel J., Wermelinger M. : A survey of self-management in dynamic software architecture specifications. In *WOSS '04 : Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems* (New York, NY, USA, 2004), ACM, pp. 28–33.

[BCK98] Bass L., Clements P., Kazman R. : *Software architecture in practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[BCO*04] Barbier F., Cauvet C., Oussalah M., Rieu D., Bennasri S., Souveyet C. : Concepts clés et techniques de réutilisation dans l'ingénierie des systèmes d'information. *Revue L'objet 10*, 1 (2004), 11–35.

[BD04] Barais O., Duchien L. : Transat : maîtriser l'évolution d'une architecture logicielle. *L'OBJET 10*, 2-3 (2004), 103–116.

[BEJV96] Binns P., Englehart M., Jackson M., Vestal S. : Domain-specific software architectures for guidance, navigation, and control. *International Journal of Software Engineering and Knowledge Engineering 6*, 2 (1996).

[Ben95] Benjamins V. : Problem solving methods for diagnosis and their role in knowledge acquisition. 93–120.

[BJC05] Batista T. V., Joolia A., Coulson G. : Managing dynamic reconfiguration in component-based systems. In *EWSA* (2005), pp. 1–17.

[BMR*96] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M. : *Pattern-oriented software architecture : a system of patterns.* John Wiley & Sons, Inc., New York, NY, USA, 1996.

[BMS08] Budinsky F., Merks E., Steinberg D. : *Eclipse Modeling Framework 2.0 (2nd Edition).* Addison-Wesley Professional, 2008.

[BMZ*05] Buckley J., Mens T., Zenger M., Rashid A., Kniesel G. : Towards a taxonomy of software change : Research articles. *J. Softw. Maint. Evol. 17*, 5 (2005), 309–332.

[BR89] Biggerstaff T. J., Richter C. : Reusability framework, assessment, and directions. 1–17.

[Bra04] Bradbury J. S. : *Organizing definitions and formalisms for dynamic software architectures.* Tech. Rep. 2004–477, Queen's University, 2004.

[Bre94] Breuker V. d. v. : *Common KADS Library for Expertise Modelling.* IOS Press, Amsterdam, The Netherlands, The Netherlands, 1994.

[CDPG*09] Chaki S., Diaz-Pace A., Garlan D., Garfunkel A., Ozkaya I. : Towards engineered architecture evolution. In *Workshop on Modeling in Software Engineering 2009* (May 2009).

[CGL*03] Clements P., Garlan D., Little R., Nord R., Stafford J. : Documenting software architectures : views and beyond. In *ICSE '03 : Proceedings of the 25th International Conference on Software Engineering* (Washington, DC, USA, 2003), IEEE Computer Society, pp. 740–741.

[CGS*02] Cheng S.-W., Garlan D., Schmerl B. R., Jo a. P. S., Spitnagel B., Steenkiste P. : Using architectural style as a basis for system self-repair. In *WICSA 3 : Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture* (Deventer, The Netherlands, The Netherlands, 2002), Kluwer, B.V., pp. 45–59.

[Cha88] Chaffin R. : The nature of semantic relations : a comparison of two approaches. 289–334.

[CHK*01] Chapin N., Hale J. E., Kham K. M., Ramil J. F., Tan W.-G. : Types of software evolution and software maintenance. *Journal of Software Maintenance 13*, 1 (2001), 3–30.

[CJS92] Chandrasekaran B., Johnson T. R., Smith J. W. : Task-structure analysis for knowledge modeling. *Commun. ACM 35*, 9 (1992), 124–137.

[Cla85] Clancey W. J. : Heuristic classification. *Artif. Intell. 27*, 3 (1985), 289–350.

[Cle96] Clements P. C. : A survey of architecture description languages. In *IWSSD '96 : Proceedings of the 8th International Workshop on Software Specification and Design* (Washington, DC, USA, 1996), IEEE Computer Society, p. 16.

[CS99] Cauvet C., Semmak F. : La réutilisation dans l'ingénierie des systèmes d'information : Etat de l'art. In *Génie Objet : analyse et conception de l'évolution d'objet.* Hermés, 1999.

[DHT01] Dashofy E. M., Hoek A. V. d., Taylor R. N. : A highly-extensible, xml-based architecture description language. In *WICSA '01 : Proceedings of the Working IEEE/IFIP Conference on Software Architecture* (Washington, DC, USA, 2001), IEEE Computer Society, p. 103.

[DI04] Dimov A., Ilieva S. : System level modeling of component based software systems. In *CompSysTech '04 : Proceedings of the 5th international conference on Computer systems and technologies* (New York, NY, USA, 2004), ACM, pp. 1–6.

[DK75] DeRemer F., Kron H. : Programming-in-the large versus programming-in-the-small. In *Proceedings of the international conference on Reliable software* (New York, NY, USA, 1975), ACM, pp. 114–121.

[DRM00] Dincel E., Roshandel R., Medvidovic N. : *ADL Independent Architectural Representation in XML.* Tech. rep., University of Southern California, May 2000.

[DSo01] DSouza D. : Model-driven architecture and integration. opportunities and challenges, March 2001.

[ER95] Euzenat J., Rechenmann F. : Shirka, 10 ans, c'est tropes ? In *LMO* (1995), pp. 13–34.

[FMvH*03] Fensel D., Motta E., van Harmelen F., Benjamins V. R., Crubezy M., Decker S., Gaspari M., Groenboom R., Grosso W., Musen M., Plaza E., Schreiber G., Studer R., Wielinga B. : The unified problem-solving method development language upml. *Knowl. Inf. Syst. 5*, 1 (2003), 83–131.

[Fow03] Fowler M. : Who needs an architect ? *IEEE Softw. 20*, 5 (2003), 11–13.

[GAO94] Garlan D., Allen R., Ockerbloom J. : Exploiting style in architectural design environments. *SIGSOFT Softw. Eng. Notes 19*, 5 (1994), 175–188.

[GAO95] Garlan D., Allen R., Ockerbloom J. : Architectural mismatch or why it's hard to build systems out of existing parts. In *ICSE '95 : Proceedings of the 17th international conference on Software engineering* (New York, NY, USA, 1995), ACM, pp. 179–185.

[Gar95] Garlan D. : What is style ? In *Proceedings of the Dagstuhl Workshop on Software Architecture* (Saarbruecken, Germany, February 1995).

[Gar00a] Garlan D. : Software architecture : a roadmap. In *ICSE '00 : Proceedings of the Conference on The Future of Software Engineering* (New York, NY, USA, 2000), ACM, pp. 91–101.

[Gar00b] Garlan D. : Software architecture and object-oriented systems. In *Proceedings of the IPSJ Object-Oriented Symposium 2000* (August 2000).

[Gar08] Garlan D. : *Evolution Styles : Formal foundations and tool support for software architecture evolution.* Tech. Rep. CMU-CS-08-142, School of Computer Science, Carnegie Mellon University, June 2008.

[GHJV94] Gamma E., Helm R., Johnson R., Vlissides J. : *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[GMF*03] Gennari J. H., Musen M. A., Fergerson R. W., Grosso W. E., Crubézy M., Eriksson H., Noy N. F., Tu S. W. : The evolution of protégé : an environment for knowledge-based systems development. *Int. J. Hum.-Comput. Stud. 58*, 1 (2003), 89–123.

[GMK02] Georgiadis I., Magee J., Kramer J. : Self-organising software architectures for distributed systems. In *WOSS '02 : Proceedings of the first workshop on Self-healing systems* (New York, NY, USA, 2002), ACM, pp. 33–38.

[GMW97] Garlan D., Monroe R., Wile D. : Acme : an architecture description interchange language. In *CASCON '97 : Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research* (1997), IBM Press, p. 7.

[Gom05] Gomaa H. : Architecture-centric evolution in software product lines. In *ECOOP-ACE '05* (Glasgow, UK, July 2005).

[GS93] Garlan D., Shaw M. : An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering* (Singapore, 1993), Ambriola V., Tortora G., (Eds.), World Scientific Publishing Company, pp. 1–39.

[GT04] Georgas J. C., Taylor R. N. : Towards a knowledge-based approach to architectural adaptation management. In *WOSS '04 : Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems* (New York, NY, USA, 2004), ACM, pp. 59–63.

[ICG*04] Ivers J., Clements P., Garlan D., Nord R., Schmerl B., Silva J. R. O. : *Documenting Component and Connector Views with UML 2.0.* Tech. Rep. CMU/SEI-2004-TR-008, Carnegie Mellon Univ., 2004.

[IEE00] IEEE Architecture Working Group : *IEEE Std 1471-2000, Recommended practice for architectural description of software-intensive systems.* Tech. rep., IEEE, 2000.

[Jaz95] Jazayeri M. : Component programming - a fresh look at software components. In *Proceedings of the 5th European Software Engineering Conference* (London, UK, 1995), Springer-Verlag, pp. 457–478.

[Jaz05] Jazayeri M. : Species evolve, individuals age. In *IWPSE '05 : Proceedings of the Eighth International Workshop on Principles of Software Evolution* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 3–12.

[JB04] Jansen A., Bosch J. : Evaluation of tool support for architectural evolution. In *ASE '04 : Proceedings of the 19th IEEE international conference on Automated software engineering* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 375–378.

[JB06] Jouault F., Bézivin J. : Km3 : A dsl for metamodel specification. In *FMOODS* (2006), pp. 171–185.

[JK05] Jouault F., Kurtev I. : Transforming models with atl. In *MoDELS Satellite Events* (2005), pp. 128–138.

[KM85] Kramer J., Magee J. : Dynamic configuration for distributed systems. *IEEE Trans. Softw. Eng. 11*, 4 (1985), 424–436.

[KOS06] Kruchten P., Obbink H., Stafford J. : The past, present, and future for software architecture. *IEEE Softw. 23*, 2 (2006), 22–30.

[Kru95] Kruchten P. : Software architecture and iterative development process. In *TRI-Ada '95 : Tutorial proceedings on TRI-Ada '91* (New York, NY, USA, 1995), ACM, pp. 491–539.

[KSO05] Khammaci T., Smeda A., Oussalah M. : *Coexistence of Object-oriented Modeling and Architectural Description*, vol. III of *Handbook of Software Engineering and Knowledge Engineering*. S. K. Chang Ed., World Scientific Publishing Co. Int, Singapore, 2005, ch. 5, pp. 119–149.

[KSO06] Khammaci T., Smeda A., Oussalah M. : Mapping cosa software architecture concepts into uml 2.0. In *ICIS-COMSAR '06 : Proceedings of the 5th IEEE/ACIS International Conference on Computer and Information Science and 1st IEEE/ACIS International Workshop on Component-Based Software Engineering,Software Architecture and Reuse* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 109–114.

[LB85] Lehman M., Belady L. (Eds.) : *Program evolution : processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[Ley04] Leymonerie F. : *ASL : un langage et des outils pour les styles architecturaux – contribution à la description d'architectures dynamiques*. PhD thesis, Université de Savoie, December 2004.

[LS80] Lientz B. P., Swanson E. B. : *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[LV95] Luckham D. C., Vera J. : An event-based architecture definition language. *IEEE Trans. Softw. Eng. 21*, 9 (1995), 717–734.

[Men02] Menzies T. : *SE/KE Reuse Research : Common Themes and Empirical Results*, vol. 2. World Scientific Publishing Co., 2002, ch. Handbook of Software Engineering and Knowledge Engineering.

[MFJ05] Muller P.-A., Fleurey F., Jézéquel J.-M. : Weaving executability into object-oriented meta-languages. In *MoDELS* (2005), pp. 264–278.

[MH01] Mencl V., Hnetynka P. : *Managing Evolution of Component Specification using a Federation of Repositories*. Tech. Rep. 2001/2, Department of Software Engineering, Charles University, Prague, June 2001.

[Min68] Minsky M. : Matter, mind, and models. In *Semantic Information Processing*, Minsky M., (Ed.). MIT Press, 1968.

[MK96] Magee J., Kramer J. : Dynamic structure in software architectures. *SIGSOFT Softw. Eng. Notes 21*, 6 (1996), 3–14.

[MKB*04] Morrison R., Kirby G., Balasubramaniam D., Mickan K., Oquendo F., Cîmpan S., Warboys B., Snowdon B., Greenwood R. M. : Support for evolving software architectures in the archware adl. In *WICSA '04 : Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture* (Washington, DC, USA, 2004), IEEE Computer Society, p. 69.

[MKM06] McVeigh A., Kramer J., Magee J. : Using resemblance to support component reuse and evolution. In *SAVCBS '06 : Proceedings of the 2006 conference*

on Specification and verification of component-based systems (New York, NY, USA, 2006), ACM, pp. 49–56.

[Mot00] Motta E. : *The Knowledge Modeling Paradigm in Knowledge Engineering*, vol. 1. World Scientific Publishing Co., 2000, ch. Handbook of Software Engineering and Knowledge Engineering.

[MR97] Medvidovic N., Rosenblum D. S. : Domains of concern in software architectures and architecture description languages. In *DSL'97 : Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997* (Berkeley, CA, USA, 1997), USENIX Association, pp. 16–16.

[MRT99] Medvidovic N., Rosenblum D. S., Taylor R. N. : A language and environment for architecture-based software development and evolution. In *ICSE '99 : Proceedings of the 21st international conference on Software engineering* (Los Alamitos, CA, USA, 1999), IEEE Computer Society Press, pp. 44–53.

[MT00] Medvidovic N., Taylor R. N. : A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng. 26*, 1 (2000), 70–93.

[MWD*05] Mens T., Wermelinger M., Ducasse S., Demeyer S., Hirschfeld R., Jazayeri M. : Challenges in software evolution. In *IWPSE* (2005), pp. 13–22.

[NR68] Naur P., Randell B. (Eds.) : *Software Engineering : Report of a conference sponsored by the NATO Science Committee*. Garmisch, Germany, October 1968.

[NR99] Nitto E. D., Rosenblum D. : Exploiting adls to specify architectural styles induced by middleware infrastructures. In *ICSE '99 : Proceedings of the 21st international conference on Software engineering* (Los Alamitos, CA, USA, 1999), IEEE Computer Society Press, pp. 13–22.

[Obj02] Object Management Group (OMG) : Meta object facility (MOF) specification. formal/2002-04-03, April 2002.

[OMK02] Oussalah M., Messaadia K., Khammaci T. : The meta modeling for reuse in kbs. In *Future Trends in Artificial Intelligence* (New Delhi, 2002), Akerkar R., (Ed.), Allied Publishers.

[OMT98] Oreizy P., Medvidovic N., Taylor R. N. : Architecture-based runtime software evolution. In *ICSE '98 : Proceedings of the 20th international conference on Software engineering* (Washington, DC, USA, 1998), IEEE Computer Society, pp. 177–186.

[OMT08] Oreizy P., Medvidovic N., Taylor R. N. : Runtime software adaptation : framework, approaches, and styles. In *ICSE Companion '08 : Companion of the 30th international conference on Software engineering* (New York, NY, USA, 2008), ACM, pp. 899–910.

[OT98] Oreizy P., Taylor R. : On the role of software architectures in runtime system reconfiguration. In *CDS '98 : Proceedings of the International Conference on Configurable Distributed Systems* (Washington, DC, USA, 1998), IEEE Computer Society, p. 61.

[Ous99] Oussalah M. : *Génie Objet : Analyse et Conception de l'Evolution.* Editions Hermès, 1999.

[Par72] Parnas D. L. : On the criteria to be used in decomposing systems into modules. *Commun. ACM 15*, 12 (1972), 1053–1058.

[PBJ98] Plásil F., Bálek D., Janecek R. : Sofa/dcup : Architecture for component trading and dynamic updating. In *CDS '98 : Proceedings of the International Conference on Configurable Distributed Systems* (Washington, DC, USA, 1998), IEEE Computer Society, p. 43.

[PDN86] Prieto-Diaz R., Neighbors J. M. : Module interconnection languages. *J. Syst. Softw. 6*, 4 (1986), 307–334.

[PW92] Perry D. E., Wolf A. L. : Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes 17*, 4 (1992), 40–52.

[RHMRM04] Roshandel R., Hoek A. V. D., Mikic-Rakic M., Medvidovic N. : Mae—a system model and environment for managing architectural evolution. *ACM Trans. Softw. Eng. Methodol. 13*, 2 (2004), 240–276.

[RR91] Royce W. E., Royce W. : Software architecture : Integrating process and technology. *TRW Quest 14*, 1 (Summer 1991), 2–15.

[SC97] Shaw M., Clements P. C. : A field guide to boxology : Preliminary classification of architectural styles for software systems. In *COMPSAC '97 : Proceedings of the 21st International Computer Software and Applications Conference* (Washington, DC, USA, 1997), IEEE Computer Society, pp. 6–13.

[SC06] Shaw M., Clements P. : The golden age of software architecture. *IEEE Softw. 23*, 2 (2006), 31–39.

[SG95] Shaw M., Garlan D. : Formulations and formalisms in software architecture. In *Computer Science Today.* Springer-Verlag, 1995, pp. 307–323.

[SG96] Shaw M., Garlan D. : *Software architecture : perspectives on an emerging discipline.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[SH07] Sadou-Harireche N. : *Evolution Structurelle dans les Architecture Logicielles à base de Composants.* PhD thesis, Université de Nantes, December 2007.

[Sha89] Shaw M. : Larger scale systems require higher-level abstractions. In *IWSSD '89 : Proceedings of the 5th international workshop on Software specification and design* (New York, NY, USA, 1989), ACM, pp. 143–146.

[Sha95] Shaw M. : Comparing architectural design styles. *IEEE Softw. 12*, 6 (1995), 27–41.

[Sha96] Shaw M. : Some patterns for software architecture. In *Pattern Languages of Program Design* (1996), vol. 2, Addison-Wesley, pp. 255–269.

[Sme06] Smeda A. : *Contribution à l'élaboration d'une métamodélisation de description d'architecture logicielle.* Thèse de doctorat, Université de Nantes, 2006.

[SMR93] Shadbolt N., Motta E., Rouge A. : Constructing knowledge-based systems. *IEEE Softw. 10*, 6 (1993), 34–38.

[SOK05] Smeda A., Oussalah M., Khammaci T. : Madl : Meta architecture description language. In *SERA '05 : Proceedings of the Third ACIS Int'l Conference on Software Engineering Research, Management and Applications* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 152–159.

[SOK08] Smeda A., Oussalah M. C., Khammaci T. : My architecture : a knowledge representation meta-model for software architecture. *International Journal of Software Engineering and Knowledge Engineering 18*, 7 (2008), 877–894.

[Som95] Sommerville I. : *Software engineering (5th ed.)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

[SS01] Sewell M. T., Sewell L. : *The Software Architect's Profession : An Introduction*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[Ste90] Steels L. : Components of expertise. *AI Mag. 11*, 2 (1990), 30–49.

[Szy98] Szyperski C. : *Component software : beyond object-oriented programming*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[Tae00] Taentzer G. : Agg : A tool environment for algebraic graph transformation. In *AGTIVE '99 : Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance* (London, UK, 2000), Springer-Verlag, pp. 481–488.

[TMA*95] Taylor R. N., Medvidovic N., Anderson K. M., E. James Whitehead J., Robbins J. E. : A component- and message-based architectural style for gui software. In *ICSE '95 : Proceedings of the 17th international conference on Software engineering* (New York, NY, USA, 1995), ACM, pp. 295–304.

[TO03] Tamzalit D., Oussalah M. : A Conceptualization of OO evolution. In *Proceedings of the 9th International Conference of Object-Oriented Information Systems (OOIS'03)* (Geneva, Switzerland, 2003), vol. 2817 of *Lecture Notes in Computer Science*, Springer, pp. 274–278.

[UO96] Urtado C., Oussalah C. : Propagation de versions dans les objets complexes. In *INFORSID* (1996), pp. 331–349.

[VCAO06] Verjus H., Cîmpan S., Alloui I., Oquendo F. : Gestion des architectures évolutives dans archware. In *CAL* (2006), pp. 41–57.

[vOvdLKM00] van Ommering R., van der Linden F., Kramer J., Magee J. : The koala component model for consumer electronics software. *Computer 33*, 3 (2000), 78–85.

[Wil99] Wile D. : Aml : An architecture meta-language. In *ASE '99 : Proceedings of the 14th IEEE international conference on Automated software engineering* (Washington, DC, USA, 1999), IEEE Computer Society, p. 183.

[Wil01a] Wile D. : Using dynamic acme. In *Working Conference on Complex and Dynamic Systems Architecture* (Brisbane, Australia, December 2001).

[WPD92] Wartik S., Prieto-Diaz R. : Criteria for comparing reuse-oriented domain analysis approaches. *International Journal of Software Engineering and Knowledge Engineering 2*, 3 (September 1992), 403–432.

Références hypertextes

[www5] *Eclipse Epsilon Plugin.*
http://www.eclipse.org/gmt/epsilon/

[www6] *Eclipse Graphical Editing Framework (GEF).*
http://www.eclipse.org/gef/

[www8] *SEI's definitions of software architecture.*
http://www.sei.cmu.edu/architecture/definitions.html

[www10] *WorldWide Institue of Software Architects (WWISA).*
http://www.wwisa.org/

Liste des tableaux

Table des figures

Table des exemples

Table des matières

Annexes

Annexe A

Référentiels de comparaison pour l'évolution basée architecture

On trouve dans la littérature d'autres référentiels de comparaison dédiés à l'évolution basée architecture que celui que nous avons proposé dans le Chapitre 2 de ce manuscrit. En particulier, notre attention a été retenue par les travaux d'Andersson en 1999, ceux de Bradbury *et al.* en 2001 et enfin ceux de Sadou *et al.* en 2007.

A.1 Référentiel d'Andersson (1999)

Dans son papier "Dimensions of Dynamism" [And99], Jasper Andersson présente son point de vue sur les architectures dynamiques en identifiant un ensemble de propriétés qui peuvent être utilisées pour décrire les caractéristiques dynamiques des architectures des applications. A cet effet, il exhibe trois dimensions du dynamisme : l'*Initiation*, le *Type*, et le *Contrôle*. Ces trois dimensions définissent le repère de la figure A.1. Ces dimensions sont détaillées dans ce qui suit.

A.1.1 Initiation du changement

Un aspect important du dynamisme architectural concerne quand et comment les tâches de modification sont initiées. Il existe deux types d'initiation : externe et interne.

- Externe : l'initiation externe est appropriée pour les reconfigurations qui ont été planifiées, par exemple pour la réactualisation de certaines parties des applications migrant vers du nouveau matériel, etc.
- Interne : des événements internes qui initient des reconfigurations peuvent être liés à l'équilibrage de charge ou d'autres exceptions dans l'application ou bien des événements extérieurs que l'application détecte, tels que des problèmes de communication et des défaillances matérielles.

A.1.2 Type du changement

Cette dimension est importante puisque c'est ici que nous pouvons voir ce qui se passe dans une architecture au moment de son exécution.

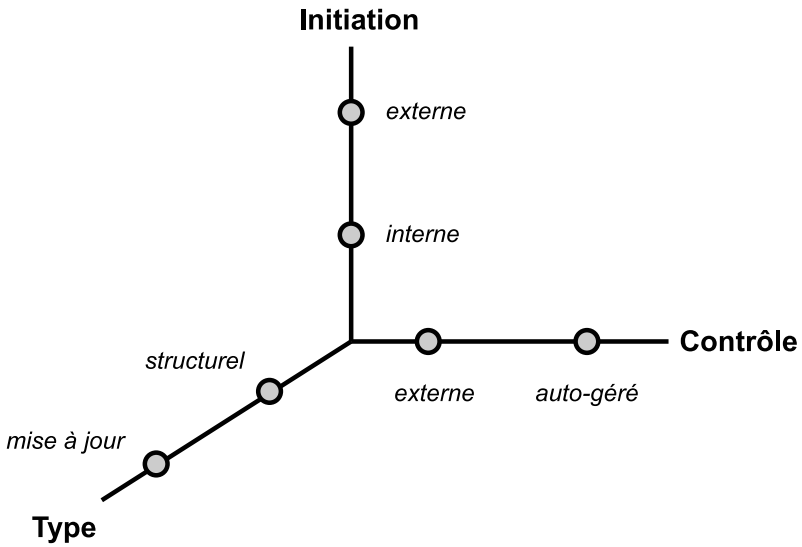

Figure A.1 – Repère tri-dimensionnel proposé par Jesper Andersson.

- Structurel : la structure d'une configuration n'est pas conservée lorsqu'une modification est appliquée à une architecture, c'est-à-dire que l'ensemble des composants, connecteurs et liaisons est affecté.
- Mise à jour : la configuration est préservée, c'est-à-dire que l'ensemble des composants et des connecteurs ainsi que l'ensemble des liaisons ne change pas. Les reconfigurations de mise à jour sont utilisées quand de nouvelles versions ou variantes de composants sont introduites dans l'application.

A.1.3 Contrôle du changement

Cette dimension détermine où le contrôle est situé. L'aspect contrôle concerne à la fois la cohérence des modifications et la réalisation des modifications en elles mêmes.

- Externe : la reconfiguration est contrôlée depuis l'extérieur. Par exemple, un dialogue s'établi entre l'application et un opérateur lui permettant d'envoyer une séquence de commandes contrôlant la reconfiguration.
- Auto géré : certaines reconfigurations, initiées en interne, sont difficilement prédictibles et plannifiables et certaines requièrent des mesures immédiates pour satisfaire les exigences du système. Ces cas demandent une approche différente où le système effectue les reconfigurations de lui même, sans interventions extérieures.

A.2 Référentiel de Bradbury *et al.* (2001)

Même si aucun référentiel n'est directement explicité, les travaux de Bradbury *et al.* [BCDW04] identifient certaines dimensions principales [1] que nous présentons sous la forme du repère à trois

1. Une version affinée de leur travail est disponible dans [Bra04]

dimensions de la figure A.2. Ces dimensions concerne l'*initiation du changement*, la *gestion du changement* et le *support d'opérations* offert pour réaliser le changement.

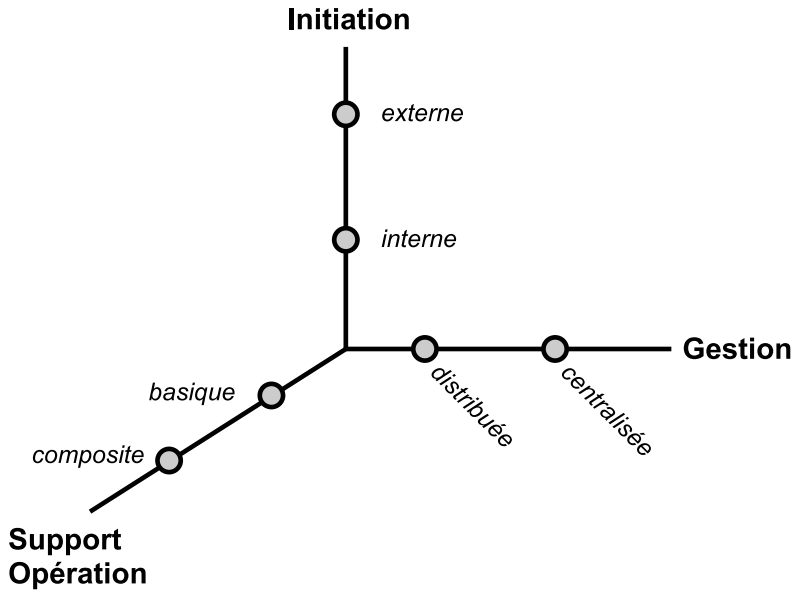

Figure A.2 – Repère tri-dimensionnel proposé par Bradbury *et al.*

A.2.1 Initiation du changement

L'initiation du changement caractérise l'endroit d'où est déclenché le changement. L'initiation peut être interne ou externe. Les auteurs définissent qu'une architecture auto-gérée est une architecture dans laquelle la globalité du processus de changement est initié en interne.

- Interne : l'initiation interne implique généralement des sondes qui fournissent les informations relatives à l'exécution sur lesquelles les décisions sont basées.
- Externe : l'initiation externe implique généralement un utilisateur extérieur, ou un système extérieur.

A.2.2 Gestion du changement

A cause de l'augmentation de la taille et de la complexité des systèmes, le passage à l'échelle est un problème important, et la gestion de la reconfiguration dans les architectures dynamiques est un facteur déterminant. En effet, on distingue la gestion centralisée et décentralisée.

- Centralisée : la gestion est centralisée dans un élément spécifique.
- Decentralisée : la gestion est distribuée à travers les composants.

A.2.3 Support d'opérations

Les changements qu'un système dynamique peut effectuer au niveau architectural sont limités par les opérations de reconfiguration disponibles. Par exemple, si un système peut uniquement ajouter des connecteurs mais pas des composants, celui-ci est limité dans sa façon de traiter les

besoins de reconfiguration. Le support se caractérise par la possibilité de spécifier des opérations de reconfiguration basiques et avancées.

- Basique : il s'agit de l'addition et de la suppression de composants et de connecteurs dans l'architecture, soit quatre opérations au total.
- Avancé : il s'agit de la combinaison des opérations basiques au sein d'un bloc structurant (*i.e.*, un groupement d'opérations) mais aussi des constructions utilisées pour les combiner (*e.g.*, séquence, choix, et itération).

A.3 Référentiel de Sadou *et al.* (2007)

Les caractéristiques utilisées pour aborder l'évolution dans [SH07] sont introduites sous formes de trois interrogations : quoi, quand et comment. Les auteurs parlent respectivement de l'*objet de l'évolution*, du *type d'évolution* et du *support de l'évolution*. Le tryptique ainsi obtenu est représenté en figure A.3. Ces dimensions sont détaillées dans ce qui suit.

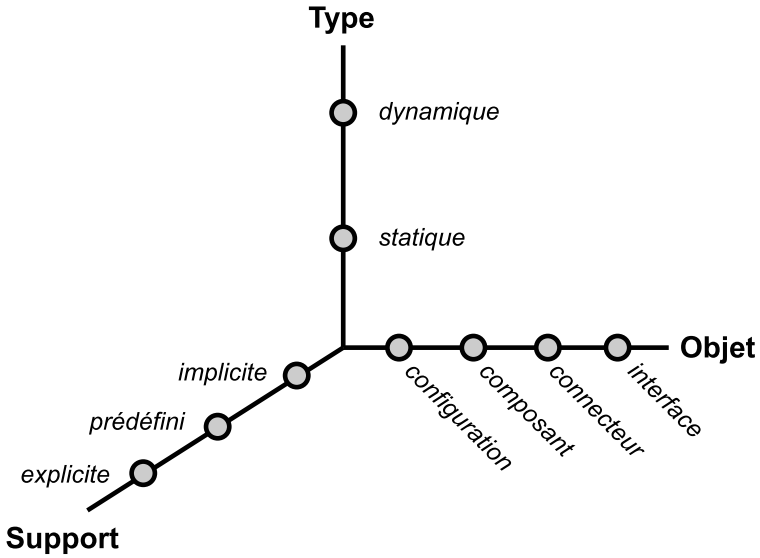

Figure A.3 – Repère tri-dimensionnel proposé par Sadou *et al.*

A.3.1 Objet de l'évolution

Cette dimension précise les éléments architecturaux sur lesquels porte l'évolution. L'objet de l'évolution peut être tout élément réifié et considéré comme citoyen de première classe par l'ADL utilisé dans la description de l'architecture concernée par l'évolution. Ainsi, une évolution peut porter sur une configuration, un composant, un connecteur, et/ou sur une interface.

A.3.2 Type d'évolution

Le type d'évolution indique à quel moment du cycle de vie est réalisée l'évolution. Cela permet de distinguer l'évolution statique de l'évolution dynamique.

– Statique : l'évolution de l'architecture logicielle est réalisée durant la phase de spécification et/ou de conception du système qu'elle décrit.
– Dynamique : l'évolution de l'architecture logicielle est réalisée durant la phase d'exécution du système qu'elle décrit.

A.3.3 Support de l'évolution

Le support d'évolution représente le moyen (opérations et mécanismes) proposé pour exprimer et gérer l'évolution d'une architecture logicielle. Il existe trois cas de figures :

– Implicite : le support de l'évolution est implicite si aucune opération d'évolution ni mécanisme ne sont offerts pour faire évoluer une architecture logicielle. Si un besoin d'évolution est apparu, alors le concepteur soit, ne sera pas en mesure de répondre à ce besoin, soit devra identifier d'une façon ad-hoc le moyen de réaliser cette évolution.
– Prédéfini : le support d'évolution est prédéfini s'il est limité à un ensemble d'opérations d'évolution et/ou de mécanismes bien précis. Il n'existe alors aucun autre moyen d'exprimer une évolution sur une architecture logicielle en dehors de ceux prédéfinis.
– Explicite : le support de l'évolution est explicite si celui-ci offre le moyen d'exprimer toute évolution que l'on souhaite appliquer sur l'architecture logicielle. Ceci implique que le support précise un ensemble d'opérations d'évolution mais aussi les mécanismes pour les réutiliser afin d'exprimer toute autre évolution sur une architecture logicielle.

A.4 Bilan

Chacun des tryptiques présentés ci-dessus offrent un angle de vue particulier sur l'évolution à l'échelle de l'architecture des systèmes logiciels. Il est à noter que certains critères peuvent parfois être recoupés et qu'un même vocabulaire peut recouvrir des significations différentes selon les auteurs. Néanmoins, ces trois référentiels offrent une complémentarité dans la compréhension de différents aspects de l'évolution architecturale. Pour terminer cette étude, nous pouvons mentionner le travail de Mens *et al.* [BMZ*05], dans lequel les auteurs proposent un référentiel se focalisant sur les détails techniques, c'est-à-dire, le "Quand", le "Où, le "Quoi" et le "Comment" des activités d'évolution. Pour répondre à ces quatre interrogations, les auteurs mettent en avant quatre dimensions respectives : les *propriétés temporelles*, l'*objet du changement*, les *propriétés système* et le *support du changement*. Toutefois, il faut bien dire que le repère à quatre dimensions ainsi obtenu reste très généraliste.

Annexe B

Algorithmes pour le raisonnement classificatoire

Au chapitre 4, nous avons montré comment un raisonnement classificatoire sur les styles d'évolution pouvait être à la base des opérations d'intégration et de recherche dans les bibliothèques de styles d'évolution. Cette annexe détaille la partie algorithmique de ce processus.

B.1 Algorithme générique de classification

Notre algorithme générique de classification est paramétré par un test de subsomption par la spécialisation ou par la composition, en vue d'être exécuté sur le contenu d'une bibliothèque de styles d'évolution (voir listing B.1). Selon la subsomption retenue, $\sqsubseteq_s$ ou $\sqsubseteq_c$, le processus de classification débutera respectivement sur les racines Evolution ou NEV. Au final, l'ensemble des SPS (les subsumants les plus spécifiques) et SPG (les subsumés les plus généraux) calculé de la sorte détermine la position d'une spécification $specif$ dans E, l'ensemble des styles d'évolution contenus dans une bibliothèque de niveau E1.

Algorithm 1 Classification

Require: $specif, root \in E$, $\sqsubseteq$ est une fonction de subsomption donnée

1: $SPS, SPG \Leftarrow \varnothing$;
2: $SPS \Leftarrow$ RechercherSPS($specif$, $root$, $\sqsubseteq$) ;
3: **for all** *entry* in SPS **do**
4: $\quad SPG \Leftarrow SPG \cup$ RechercherSPG($specif$, $entry$, $\sqsubseteq$)
5: **end for**

On remarquera dans le listing B.1 que le calcul des SPS sert de point d'entrée au calcul des SPG. Les fonctions *RechercherSPS* et *RechercherSPG* sont détaillées dans les sections qui suivent.

B.2 Algorithme de recherche des SPS

Le principe de l'algorithme du listing B.2 est de parcourir la hiérarchie en profondeur en partant de la racine jusqu'à trouver une description de style qui ne subsume plus la description

à classer x. Le squelette de cette fonction récursive est donné ci-dessous, où *current* représente le style d'évolution courant.

Algorithm 2 RechercherSPS

Require: $x, current \in E$, $\sqsubseteq$ est une fonction de subsomption

```
1: L ← ∅ ;
2: if not x ⊑ current then
3:    return ∅ ;
4: else
5:    if isLeaf(current, ⊑) then
6:       return current ;
7:    else
8:       L ← ∅ ;
9:    end if
10:   for all d in descendants(current, ⊑) do
11:      L ← L ∪ RechercherSPS(x, d, ⊑) ;
12:   end for
13:   if L = ∅ then
14:      return current ;
15:   else
16:      return L ;
17:   end if
18: end if
```

B.3 Algorithme de recherche des SPG

Il suffit de ne considérer que l'ensemble des descendants du SPS. Si un des descendants est subsumé par la description du style à classer x, alors il est un SPG et sa descendance est ignorée, sinon leurs descendants sont testés à leur tour jusqu'à ce qu'un SPG soit trouvé ou qu'il n'y ait plus aucun descendants à tester. L'algorithme correspondant est présenté dans le listing B.3.

B.4 Remarques et améliorations

Dès que les éléments sont structurés hiérarchiquement, la même technique de classification peut être employée. C'est tout l'intérêt de l'algorithme générique exposé dans cette section. En revanche, la détermination du subsumant et du subsumé est fondamentalement différente selon que l'on considère la relation de spécialisation ou de composition. Le test de subsomption de spécialisation s'appuie sur les paramètres et les relations de composition et d'utilisation d'un entête. Le test de subsomption de composition s'appuie uniquement sur les relations de composition d'un entête. Aussi, les tests de subsomption par la spécialisation et par la composition sont des fonctions qui utilisent les styles comme des boites noires, ignorant leurs détails d'implémentation.

La technique de classification en elle même peut être améliorée suivant différentes directions.

Algorithm 3 RechercherSPG

Require: $x, current \in E$, $\sqsubseteq$ est une fonction de subsomption

```
1: L ← ∅ ;
2: if current ⊑ x then
3:    return current
4: else
5:    if isLeaf(current, ⊑) then
6:       return ∅ ;
7:    else
8:       for all d in descendants(current,⊑) do
9:          L ⇐ L ∪ RechercherSPG(x, d, ⊑) ;
10:      end for
11:      return L
12:   end if
13: end if
```

B.4.1 Optimisation de la traversée

Dans le cas de la composition mais aussi dans le cas où l'héritage multiple serait autorisé, les hiérarchies obtenues sont de type treillis. La conséquence directe de cela est que certains noeuds de la hiérarchie peuvent être visités plusieurs fois lors de l'exécution de l'algorithme de classification. Pour remédier à cela, on peut recourir à un système de marquage afin d'éviter de re-visiter des noeuds.

B.4.2 Hiérarchies orthogonales

Dans l'approche choisie, nous disposons de deux hiérarchies distinctes et bien définies, enracinées à `Evolution` et `NEV`. Une autre approche consiste à considérer ces deux hiérarchies comme étant orthogonales l'une de l'autre. Le principe étant de considérer qu'il n'existe pas qu'une seule hiérarchie de composition, mais plusieurs hiérarchies, enracinées différemment. Ainsi, chaque style d'évolution est potentiellement la racine d'une hiérarchie de composition. Le cœur de l'algorithme générique de classification reste inchangé, mais la phase d'initialisation relative à la composition est modifiée dans la mesure où il faut pouvoir déterminer les multiples racines. Dans ce cas de figure, il est judicieux d'explorer la hiérarchie de spécialisation d'abord, en considérant le style d'évolution courant comme la racine potentielle d'une hiérarchie de composition qu'il convient alors d'explorer.

B.4.3 Mesure du degré de subsomption

Afin d'affiner le raisonnement classificatoire, il est intéressant de s'appuyer sur une distance d'inclusion – ou degré de subsomption. Il s'agit d'une mesure asymétrique entre deux descriptions de styles d'évolution et dont le résultat va permettre de restreindre le nombre de SPS et de SPG. En effet, dans l'approche choisie, le test de subsomption est une fonction binaire, c'est-à-dire soit il réussi, soit il échoue. Mais on peut décider de mesurer le degré de subsomption, exprimé dans l'intervalle [0..1]. Au dessous d'un certain seuil (fixé empiriquement), des SPS/SPG ne seront pas retenus lors de l'exploration de la hiérarchie. On trouve dans la littérature plusieurs techniques

destinées à calculer le degré d'inclusion d'un concept dans un autre, caractérisant la distance et ainsi la similarité entre deux concepts. Cette technique serait particulièrement intéressante pour l'opération de recherche dans la bibliothèque afin de sélectionner les meilleurs styles candidats à la réutilisation en fonction des besoins exprimés par l'utilisateur.

B.4.4 Traitement sémantique

Le raisonnement classificatoire opéré ici peut parfois être considéré comme erroné du point de vue sémantique. Ceci est dû au fait que l'algorithme de structuration taxinomique repose sur un traitement purement syntaxique. En l'occurrence, ce sont les informations syntaxiques de la partie entête d'un style qui entrent en jeu (*i.e.*, paramètres et relations), tandis que les informations sémantiques ne sont pas exploitées (*i.e.*, nom du style et description textuelle du but). Il est parfaitement envisageable d'utiliser un thésaurus afin de guider le raisonnement classificatoire. Par exemple, on pourrait inférer une relation entre deux styles d'évolution si il existe une relation de synonymie entre le nom de ces derniers. La convention de nommage des styles d'évolution, de la forme Evolution@ElementEvolutif facilitera ce traitement dans la mesure ou l'opérateur et l'opérande sont facile à isoler.

Annexe C

Le prototype COSAStudio

Cette annexe concerne le développement d'un outil CASE[1] baptisé COSAStudio. L'objectif du projet COSAStudio consiste à proposer un prototype pour la conception et l'évolution d'une architecture logicielle. Face à la difficulté d'évaluer quantitativement les propositions liées à cette thèse, COSAStudio permet de concrétiser un certain nombre d'entre elles.

C.1 Présentation générale de COSAStudio

Dans leur article [JB04], Anton Jansen et Jan Bosch concluent leur évaluation sur le fait qu'en général, les outils d'architecture ne voient pas l'évolution comme une partie inhérente et une dimension séparée de l'architecture logicielle. En effet, dans la majorité des outils (ArchJava, ArchStudio, AcmeStudio, etc.), l'accent est mis sur la définition de l'architecture de façon correcte. L'évolution de l'architecture est souvent supervisée, non implémentée, et laissée à la charge d'outils idoines. Par conséquent, l'outillage pour l'évolution architecturale n'est actuellement réalisée que partiellement. L'outil COSAStudio vise à permettre aussi bien aux architectes de décrire leurs architectures que de les faire évoluer.

Le développement d'un nouvel outil de modélisation est une tâche difficile et chronophage. Pour accélérer et faciliter cette tâche, COSAStudio est construit à partir de l'environnement de modélisation d'Eclipse, l'*Eclipse Modelling Framework* (EMF).

C.2 Eclipse Modelling Framework

EMF (Eclipse Modeling Framework) est un cadre de modélisation pour Eclipse. Le cœur du framework EMF inclue un méta-méta-modèle (Ecore) pour décrire des modèles et un support exécutable pour les modèles, intégrant un mécanisme de notification des changements, un support pour la persistance à travers une sérialisation XMI, et une API réflexive pour manipuler des objets EMF de manière générique. En d'autres termes, Ecore définit la structure des méta-modèles qui définissent à leur tour les modèles que les développeurs utilisent pour conserver les données de leurs applications. EMF peut être vu comme comme une implémentation Java efficace du coeur de l'API du MOF de l'OMG. Cependant, afin d'éviter toute confusion, le méta-méta-modèle dans EMF se nomme Ecore. Dans la proposition actuelle de MOF 2.0, un sous-ensemble du MOF, appelé EMOF (Essential MOF), est séparé. Il y a de légères différences, principalement

1. Acronyme anglais pour *Computer Aided Software Engineering*.

de nommage, entre Ecore et EMOF. Les principaux concepts de Ecore sont présentés dans la Figure C.1.

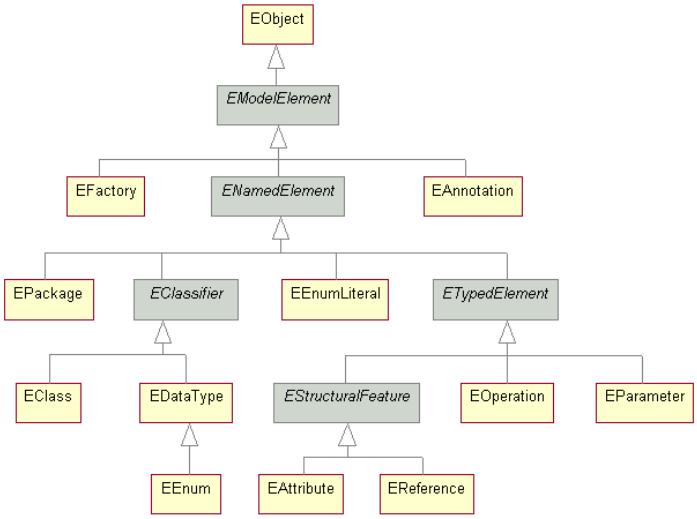

Figure C.1 – Hiérarchie des concepts de Ecore.

Du point de vue de la méta-modélisation, EMF supporte une hiérarchie à trois niveaux : le niveau méta-méta-modèle (Ecore), le niveau méta-modèle et le niveau modèle. Au sommet de la hiérarchie, Ecore est auto-défini, c'est-à-dire que l'on trouve dans EMF le fichier `ECORE.ecore`. Puis, à l'aide de ce dernier, on peut définir par exemple le méta-modèle UML 2.0 par le fichier `UML2.ecore`. A la base de la hiérarchie de modélisation, l'utilisateur final peut se créer un diagramme au choix d'UML dans le fichier `diag1.uml`. Dans EMF, l'extension donnée aux fichiers (*i.e.*, les modèles sérialisés en XMI) reflète le nom du méta-modèle et donc du langage qui a servi à les décrire.

C.3 Éléments caractéristiques de COSAStudio

Cette section donne un aperçu des choix et solutions techniques retenues pour COSAStudio et qui peuvent probablement être réutilisées pour la construction d'autres outils de ce genre.

C.3.1 Interface graphique utilisateur

COSAStudio a fait le choix d'un environnement de modélisation entièrement visuel où chaque artefact possède une représentation graphique, même minimaliste. On peut schématiser son interface graphique par la Figure C.2. Sa structuration est héritée de celle d'Eclipse. Les différentes parties de la structure sont :

- Menu : donne accès à toutes les actions possibles de l'outil. Cette barre peut être adaptée en fonction du contexte (*i.e.*, si aucun modèle n'est en cours d'édition, il est inutile d'afficher les actions associées).
- Explorateur : il est nécessaire de pouvoir organiser (supprimer, déplacer, renommer, etc.) les différentes ressources produites ou consommées durant l'activité de modélisation.

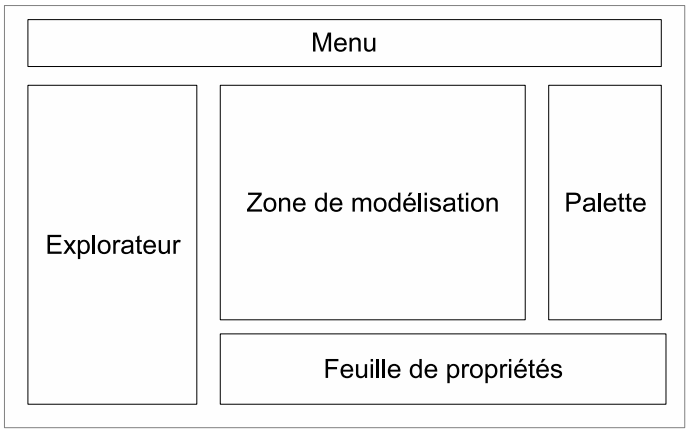

Figure C.2 – Modèle d'interface utilisateur pour l'outil COSAStudio.

- Zone de modélisation : c'est l'éditeur de modèles. Il permet de visualiser et de manipuler les éléments de modélisation. Par défaut, il se charge de la création, des modifications et des suppressions de ces éléments. Une représentation arborescente des éléments est offerte par défaut. Si Eclipse GEF (*Graphical Editing Framework*) [WWW] est exploité, les éléments bénéficient d'une représentation graphique personnalisée.
- Feuille de propriétés : permet d'afficher et d'éditer les propriétés des éléments de modélisation. Cette feuille est accessible lorsqu'un élément du modèle est sélectionné.
- Palette : propose les outils permettant l'interaction entre l'utilisateur et l'éditeur. Il s'agit d'outils de création d'éléments par "glisser-déposer", de zoom, de sélection. Son apparence doit être définie par GEF.

C.3.2 Hiérarchie de modélisation

La Figure C.3 donne une vision d'ensemble des différents modèles intégrés dans le fonctionnement de COSAStudio, dans le cadre du framework EMF. Au niveau M2, on trouve les méta-modèles de COSA, de SAEM, d'UML et du langage de transformation ATL [JK05] (*Atlas Transformation Language*). Au niveau M1 on trouve les modèles `TU.cosa` et `TUEvol.saem`. Le premier contient la définition des types COSA pour le domaine de la tuyauterie. Le second contient les styles d'évolution associés à ces types. On trouve également un ensemble de transformations utiles.

EMF n'accepte que les trois niveaux de modélisation M3, M2 et M1. Or, c'est au niveau M0 (ou A0 dans le paradigme des architectures) que l'utilisateur final doit décrire son application, en l'occurrence `TU20.tu`. Nous expliquons dans la section suivante comment introduire dans EMF le dernier niveau.

C.3.3 Introduction du niveau A0

Le framework EMF est basé sur Java, et de ce fait est enraciné dans une dichotomie type–instance. En effet, le langage Java n'accepte que deux niveaux de modélisation effectif : les types (*i.e.*, les classes java) et les instances (*i.e.*, les objets java). Ceci suffit à supporter nativement les niveaux de modélisation M3 et M2. Pour obtenir le troisième niveau M1, EMF utilise une

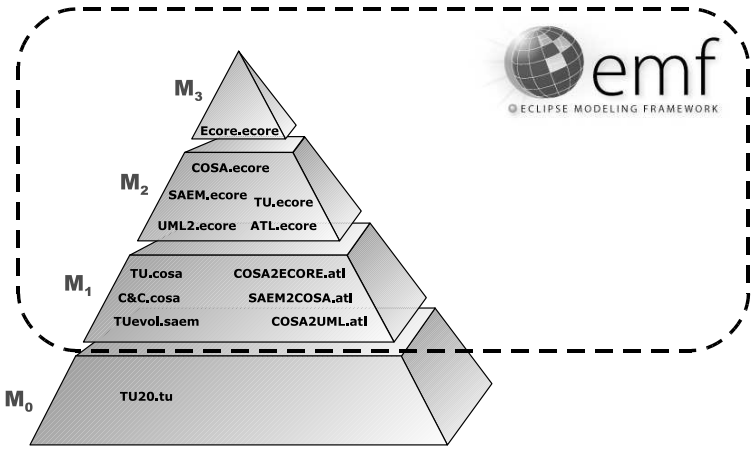

Figure C.3 – Vue d'ensemble des différents modèles nécessaires à COSAStudio.

technique de DataBinding (production des classes Java à partir de document XMI) ou encore son API réflexive. L'introduction d'un quatrième et dernier niveau M0 consiste à ré-appliquer ces principes par simple décalage des niveaux de modélisations considérés. Ainsi, la transformation représentée sur la Figure C.4 vise à "remonter" le langage TU au niveau M2 afin de le poser comme base d'un nouvel environnement de modélisation. De ce point de vue, COSAStudio peut être perçu comme un méta-outil de modélisation.

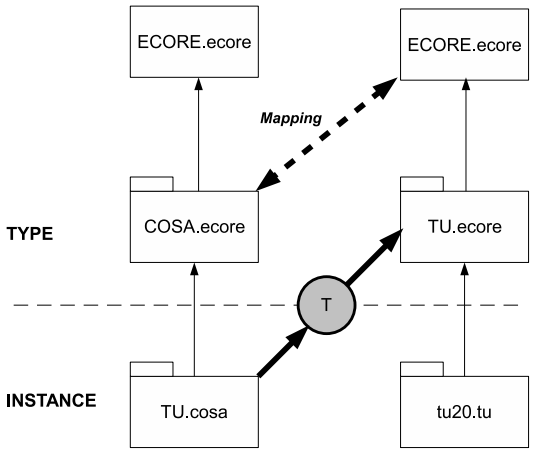

Figure C.4 – Introduction d'un niveau de modélisation M0 par une technique de transformation de modèle.

Le méta-modèle d'entrée de la transformation est COSA (`COSA.ecore`), et le méta-modèle de sortie est Ecore (`Ecore.ecore`). La description de la transformation en ATL repose sur les correspondances entre les concepts des deux méta-modèles. Selon les règles de correspondances établies dans `COSA2ECORE.atl`, le résultat de la transformation de `TU.cosa` est `TU.ecore`.

C.4 Les perspectives dans COSAStudio

Dans la plate-forme Eclipse, il y a deux couches principales : la couche modèle et la couche interface utilisateur. Le modèle sous-jacent, connu sous le nom de Workspace, est un ensemble de ressources (projets, dossiers et fichiers). L'interface utilisateur, ou Workbench, définit la présentation de ces ressources. Dans le Workbench, la fonctionnalité *perspective* est utilisée pour contrôler la visibilité des éléments dans le modèle et dans l'interface utilisateur. Elle contrôle ce qui est visible dans le modèle (quel projet, dossier ou fichiers) et ce qui est visible dans l'interface utilisateur (quelles actions ou vues). Ces contrôles permettent de naviguer dans le Workspace et de le modifier d'une manière qui corresponde aux objectifs des utilisateurs. Étant construit sur la plateforme Eclipse, COSAStudio suit les mêmes principes. Nous avons définis trois perspectives différentes : COSA, C&C et SAEM.

C.4.1 La Perspective COSA

La perspective COSA permet à l'architecte métier de définir ses types d'éléments de modélisation architecturaux en COSA. Il définit ses vues (tuyauterie, gaine, électricité, ...) comme des ensembles bien définis de types. Il a à sa disposition différents éléments de modélisation : type de configuration, type de composant, type de connecteur, etc., destinés à être regroupés dans un élément de modélisation racine : la bibliothèque de types. La modélisation se fait par instanciation des concepts contenus dans le fichier `COSA.ecore`. Une capture d'écran de la perspective COSA dans l'outil COSAStudio est visible sur la Figure C.5.

C.4.2 La Perspective C&C

A première vue, on s'attend à ce que les éléments de modélisation de chaque métier offrent un vocabulaire et une notation graphique spécifique. Mais ce résultat nécessite une phase de personnalisation préalable, où chaque méta-modèle est associé à une nouvelle perspective et un nouvel aspect graphique. Pour pouvoir se passer de cette personnalisation qui reste manuelle et fastidieuse, COSAStudio offre à la place une perspective minimale et commune C&C, où les éléments de modélisation standard sont le composant et le connecteur. Toutefois, l'aspect métier est pris en compte lors de leur intégration dans le modèle, où leur véritable nature est sélectionnée. Par exemple, comme le montre la Figure C.6, plutôt que de créer les perspectives Tuyauterie, Gaine et Electricité, la perspective C&C suffit. La modélisation se fait par instanciation des concepts contenus dans le fichier correspondant à la vue métier désirée (`TU.ecore`, `Gaine.ecore`, `Elec.ecore`, ...). Une capture d'écran de la perspective C&C dans l'outil COSAStudio est visible sur la Figure C.6.

C.4.3 La Perspective SAEM

C'est dans cette perspective que sont modélisés les styles d'évolution. Pour ce faire, l'architecte d'application évolutive peut instancier les concepts défini dans le méta-modèle `SAEM.ecore` et les associer aux concepts de la vue métier désirée (`TU.ecore`, `Gaine.ecore`, `Elec.ecore`, ...). Ainsi, l'AAE peut construire des bibliothèques de niveau E1 adaptées à chaque vue métier. Grâce à EMF.Edit, l'architecte dispose d'un éditeur arborescent pour décrire son modèle SAEM. Nous n'avons pas défini de notation graphique personnalisée puisque ces modèles sont destinés à être

projetés vers COSA et ainsi de bénéficier de la notation de ce dernier. Une capture d'écran de la perspective SAEM dans l'outil COSAStudio est visible sur la Figure C.7.

Dans cette perspective, nous avons implémenté l'opération d'intégration d'une nouvelle description dans la bibliothèque (selon le point de vue spécialisation pour l'instant). Le lien entre le code java de l'algorithme de classification et son intégration sous forme d'un menu contextuel sous COSAStudio a été rendu possible par une technique de script EWL inclus dans le plugin Epsilon [WWW].

C.5 Améliorations futures

Le développement du prototype COSAStudio valide le travail réalisé par Smeda *et al.* sur l'ADL COSA [Sme06]. Le prototype valide également une partie des idées proposées dans cette thèse autour du paradigme des styles d'évolution. Le problème est que le framework EMF facilite la manipulation d'entités de type produit, mais beaucoup moins celle d'entités de type processus comme c'est le cas des styles. A l'heure actuelle, la perspective SAEM permet à un architecte d'applications évolutives de modéliser correctement des styles d'évolution, mais il nous a été très difficile d'incorporer dans COSAStudio l'aspect exécutable des styles d'évolution. Cette fonctionnalité est une priorité pour la future version de l'outil.

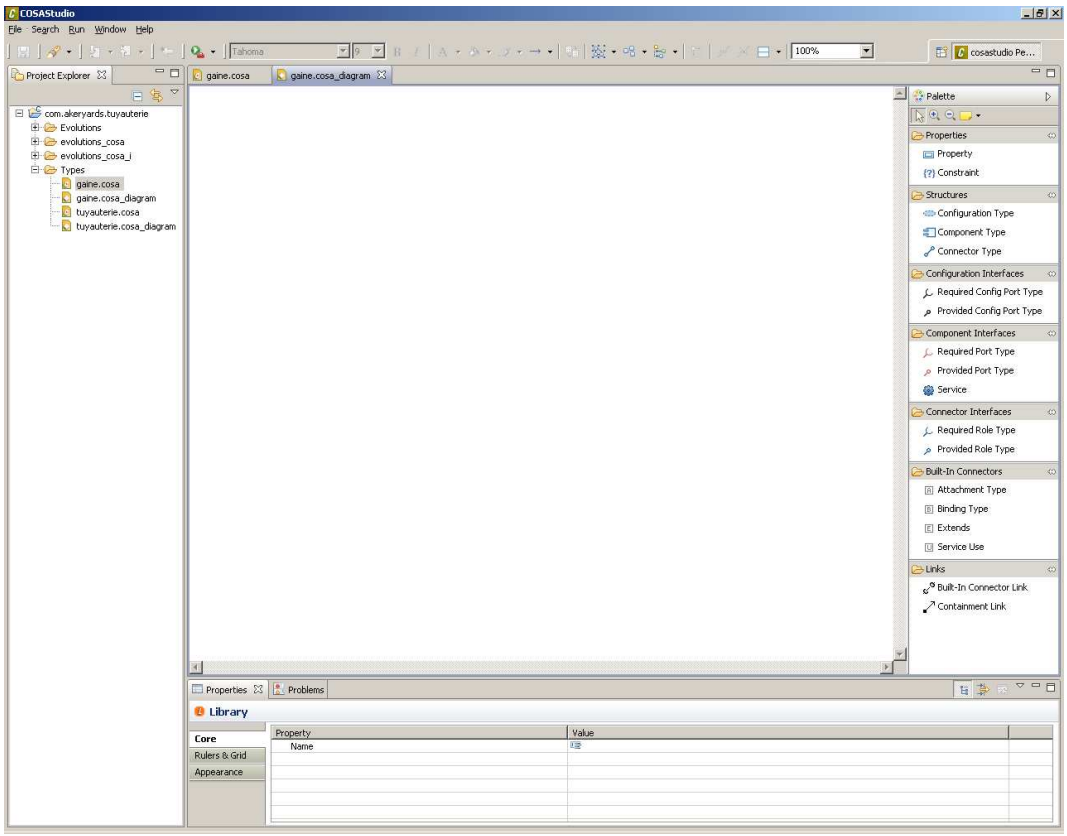

Figure C.5 – Perspective COSA sur un modèle vierge.

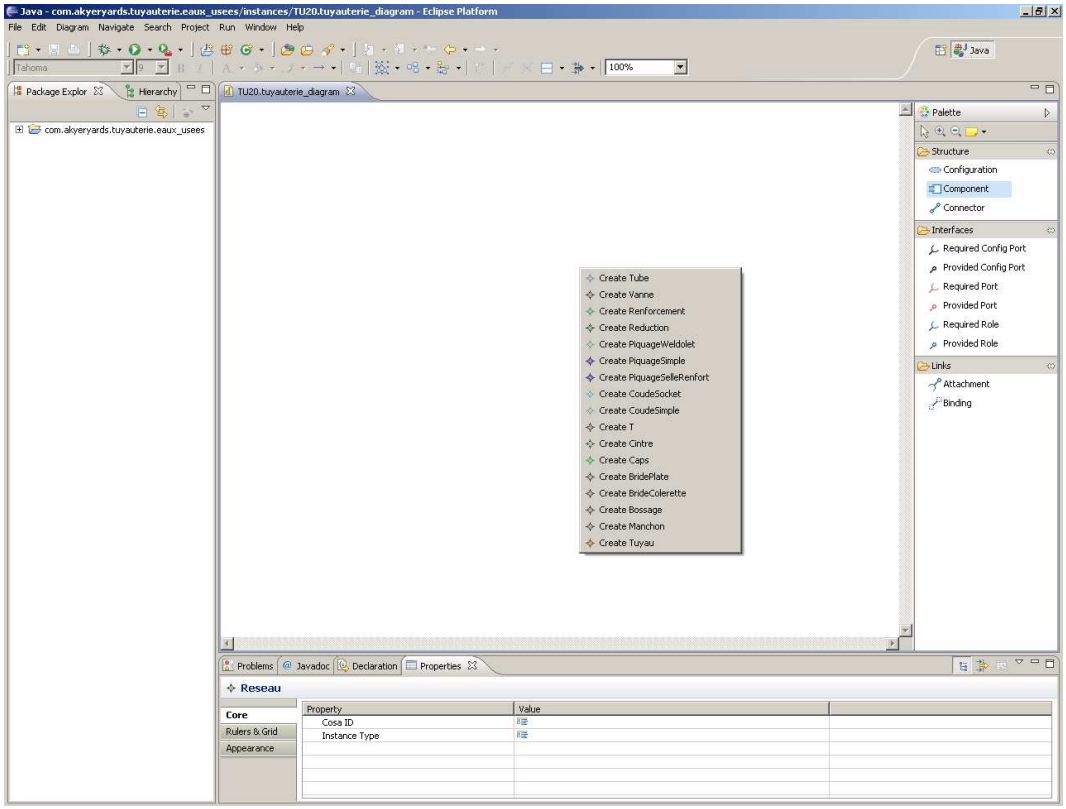

Figure C.6 – Perspective C&C sur un modèle vierge pour le métier de la tuyauterie.

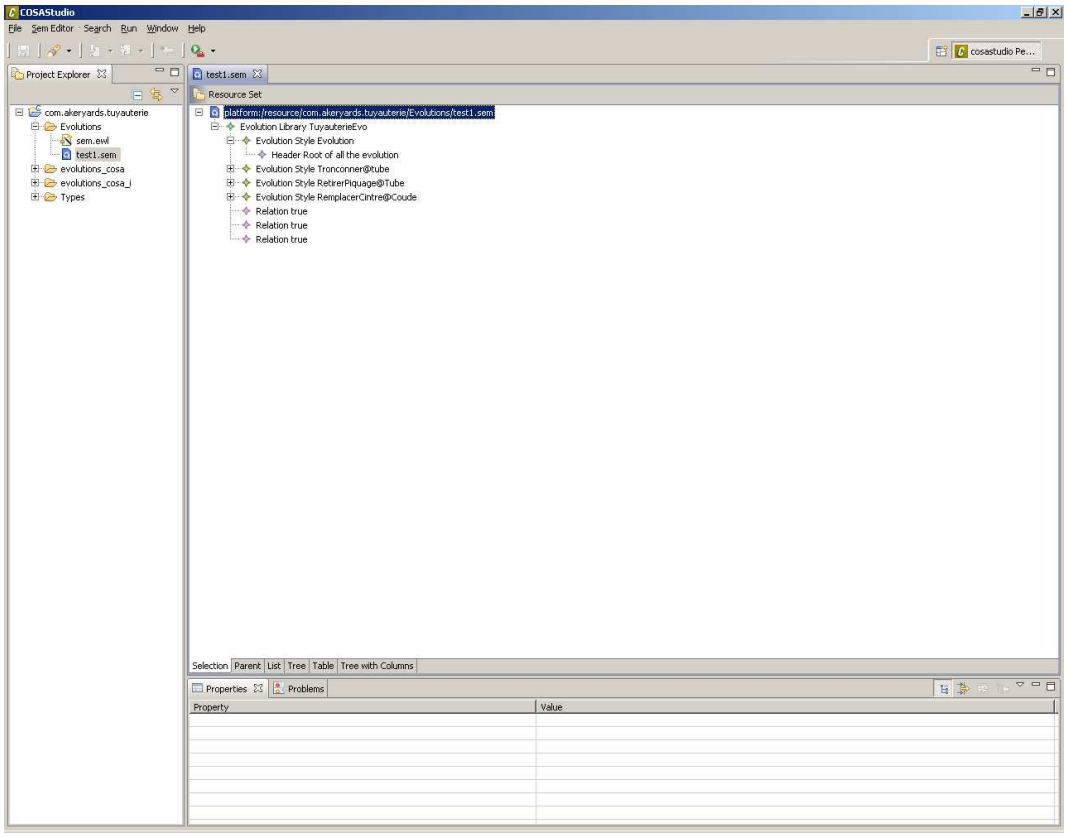

Figure C.7 – Perspective SAEM sur un modèle vierge.

Styles d'évolution dans les architectures logicielles

Olivier Le Goaer

Résumé

Les architectures logicielles ont été introduites en réponse à l'accroissement de la complexité des systèmes, en favorisant leurs descriptions à un haut niveau d'abstraction. Dans cette thèse, nous proposons d'aborder la problématique de leurs évolutions avec comme objectif, de capitaliser les évolutions récurrentes et de favoriser leur réutilisation. Notre contribution se décline en deux volets majeurs. Le premier volet concerne la proposition du modèle d'évolution SAEM (*Style-based Architectural Evolution Model*), permettant l'abstraction, la spécification et la gestion de l'évolution dans les architectures logicielles au travers du concept de style d'évolution. SAEM se veut un modèle d'évolution générique, uniforme et indépendant de tout langage de description d'architecture. Le formalisme proposé décrit les concepts du style d'évolution selon un tryptique : domaine, entête et compétence. Le deuxième volet concerne le développement d'une approche de réutilisation par dessus SAEM pour tenter de rendre les activités d'évolution plus rentables. Nous proposons une démarche pour la construction de bibliothèques pour les styles d'évolution, orchestrée par différentes catégories d'intervenants. Les bibliothèques sont élaborées selon deux types de processus complémentaires : « pour la réutilisation » et « par la réutilisation ». Nous présentons une technique de raisonnement classificatoire pour permettre aux bibliothèques d'être peuplées et interrogées dans le but de gérer les savoir et savoir-faire relatifs à l'évolution architecturale.

Mots-clés : Style d'évolution, Patron d'évolution, Réutilisation de l'évolution, Bibliothèque, Architecture Logicielle, Langage de Description d'Architecture.

Abstract

Software architectures have been introduced in response to the increasing complexity of systems, by leveraging their descriptions at a high level of abstraction. In this thesis, we propose to tackle the problem of their evolutions with the aim of capitalizing the recurrent evolution and fostering their reuse. Our contribution is divided into two major parts. The first part concerns with proposing the evolution model SAEM (Style-based Architectural Evolution Model), allowing abstraction, specification and management of evolution in software architectures through the concept of evolution style. SAEM is a generic evolution model, consistent and independent of any architectural description language. The proposed formalism describes evolution style's concepts according to a triptych: domain, header and competence. The second part concerns with the development of a reuse-driven approach on top of SAEM to try to make evolutions more cost-effective activities. We propose an approach for the construction of libraries dedicated to evolution styles, orchestrated by several stakeholders. Libraries are developed with two complementary types of processes: « for reuse » and « by reuse ». We explain a classification-based reasoning technique to enable libraries to be enriched and queryied in order to manage the knowledge and know-how related to architectural evolution.

Keywords: Evolution Style, Evolution Pattern, Evolution Reuse, Library, Software Architecture, Architecture Description Language.